AD-A041 776 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
IMPROVING PROGRAMS BY SOURCE-TO-SOURCE TRANSFORMATION.(U)
APR 77 P E RUTTER DAAB07-72-C-0259
UNCLASSIFIED R-765 NL

1 OF 2
ADA041776

REPORT R-765 APRIL, 1977 UILU-ENG 77-2212

CSL COORDINATED SCIENCE LABORATORY

12

IMPROVING PROGRAMS BY SOURCE-TO-SOURCE TRANSFORMATION

PAUL EDWARD RUTTER

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

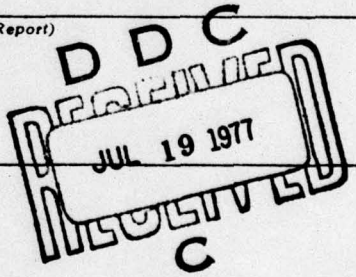
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) IMPROVING PROGRAMS BY SOURCE-TO-SOURCE TRANSFORMATION.		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report.
		6. PERFORMING ORG. REPORT NUMBER (14) R-765, UILU-ENG-77-2212
7. AUTHOR(s) (10) Paul Edward Rutter		8. CONTRACT OR GRANT NUMBER(s) (15) DAAB-07-72-C-0259, N00014-75-C-0612
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		(11) 12. REPORT DATE April, 1977 13. NUMBER OF PAGES 109
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 118 p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)



18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Source-to-Source
Automatic Program Improvement
Program Manipulation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes a program manipulation system which speeds up Lisp programs by rewriting them. The system can read in programs and input test sets, run the program in a controlled fashion, and propose and carry out changes to the program which may result in faster execution and better algorithms. Improvements fall into two categories: those based solely on the program text, and those which use information learned from running the program. Changes are tested for correctness (program gives the same answers) and effectiveness (actually speeds the programs up).

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

OVER

097700

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

Discussed in the thesis are the programmable language-oriented editor which carries out transformation commands, the library of patterns used to organize program transformations, a data and control flow analysis routine, and tools for investigating the dynamic behavior of programs. One section is devoted to a discussion of the controversy surrounding the area of program proving, and gives arguments in support of the programmer-aided program testing techniques used in the transformation system.

The system was applied to a number of programs, ranging in size from single functions to large programs twenty-five pages in length. The improved programs had execution speeds ten to fifty per cent faster than the originals.

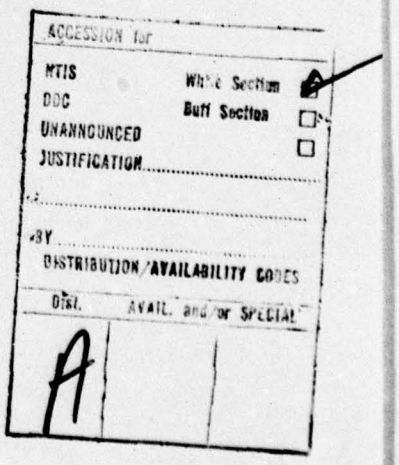
UILU-ENG 77-2212

IMPROVING PROGRAMS BY SOURCE-TO-SOURCE TRANSFORMATION

by

Paul Edward Rutter

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and in part by the Office of Naval Research under Contract N00014-75-C-0612.



IMPROVING PROGRAMS BY SOURCE-TO-SOURCE TRANSFORMATION

BY

PAUL EDWARD RUTTER

S.B., Massachusetts Institute of Technology, 1972

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor Robert T. Chien

Urbana, Illinois

IMPROVING PROGRAMS BY SOURCE-TO-SOURCE TRANSFORMATION

Paul Edward Rutter, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1977

This thesis describes a program manipulation system which speeds up Lisp programs by rewriting them. The system can read in programs and input test sets, run the program in a controlled fashion, and propose and carry out changes to the program which may result in faster execution and better algorithms. Improvements fall into two categories: those based solely on the program text, and those which use information learned from running the program. Changes are tested for correctness (program gives the same answers) and effectiveness (actually speeds the program up).

Discussed in the thesis are the programmable language-oriented editor which carries out transformation commands, the library of patterns used to organize program transformations, a data and control flow analysis routine, and tools for investigating the dynamic behavior of programs. One section is devoted to a discussion of the controversy surrounding the area of program proving, and gives arguments in support of the programmer-aided program testing techniques used in the transformation system.

The system was applied to a number of programs, ranging in size from single functions to large programs twenty-five pages in length. The improved programs had execution speeds ten to fifty per cent faster than the originals.

ACKNOWLEDGMENTS

I wish to thank my research advisor, Professor Dave Waltz, for his support and encouragement during the development of this thesis. I also thank Professor R. T. Chien and all the members of the Artificial Intelligence group at the Coordinated Science Lab for providing an interesting research environment.

TABLE OF CONTENTS

I

INTRODUCTION

This thesis describes a program manipulation system which speeds up Lisp programs by rewriting them. The system can read in programs and input test sets, run the program in a controlled fashion, and propose and carry out changes to the program which may result in faster execution and better algorithms. Improvements fall into two categories: those based solely on the program text (static match-replacements), and those which use information learned from running the program. Changes are tested for correctness (program gives the same answers) and effectiveness (actually speeds the program up).

Also described are various tools constructed for the system which have proved useful in their own right, including a programmable Lisp editor, a calling tree generator, and several metering and monitoring programs.

I.1 The Need for Optimization

The need for some form of automatic optimization has been clear from the first days of higher level languages. The original reason for higher level languages was to make programming easier, and a large part of that task involves hiding the "ugly realities" and eccentricities of the physical machine from the programmer. At the same time, the intricacies of machine language and the enormous number of possible machine code sequences to do the same task means that the language translator will often choose a sequence which is not the fastest. Even worse are the inefficiencies suffered due to the modular approach taken by most language translators (which is very worthwhile for other reasons). Separate compilation of functions and standardized module interfaces make for uniform code, but they often lead to redundant computation (for example, saving all the registers before every subroutine call, even though many subroutines do not use all the registers). Much of the early work on compiler optimization was aimed at removing this sort of "ridiculous" code [Nievergelt], and making the source code efficiently reflect the program written in the higher level language.

In addition to "machine level" optimizations, perhaps typified by the problem of register optimization, most modern compilers do some other optimizations which really involve <u>not</u> following the programmer's prescription exactly. Some examples are eliminating the redundant

computation of common subexpressions, compile time computation of constants, and moving invariant computations out of loops [Allen 71].

With the introduction of ever "higher level" languages and automatic programming systems, the problem of optimization takes on a central role. It has become far easier to produce a working program, but it is also easier to produce an inefficient one [Cheatham 72]. Very high level languages allow the concise specification of algorithms dealing with complicated data structures. In these languages it is often easy to write an inefficient algorithm without even realizing that one has done so, because so many operators are built-in. In a set-oriented language for example, the easiest-to-write algorithm may unnecessarily generate many intermediate copies of the sets involved [Schwartz]. The question of optimization may also involve choosing a good underlying machine representation for the sets, which in general depends on the particular program being run and thus can not be built into the system ahead of time [Low]. One purpose of an automatic optimizer then, is to let us "have our cake and eat it too." Within the last few years there has been increasing interest in extending the power of automatic optimizers even further -- to the point where they can come up with significant improvements to the actual algorithm [Darlington] [Wegbreit 76] [Standish]. This has proved quite difficult for useful size programs -- often because it can be very difficult to separate the "algorithms" from the "incidentals" in real programs. Nonetheless, the idea of having an "algorithm assistant" is very exciting, and shows

promise. Another area of optimization is that of specializing the program for the data, or dynamic analysis. This involves actually running the program in order to observe its execution behavior and gather statistics, and then modifying the program to take advantage of any "lopsidedness." Running the program also gives many clues as to how the algorithm might be improved.

In this thesis, I will try and point out the virtues and practicality of doing most of the optimizations mentioned above within the source language. There are three main advantages to optimizing within the source language. The first is that the programmer need only know one language -- the output may be a little more unreadable than the input, but it is at the same conceptual level, in the same syntax, and for the same "machine." The second advantage is less obvious: with source-to-source transformations, the program can be cycled through the system more than once, feeding the output back to the input. This makes much of the system design easier since it allows one to worry about one type of optimization at a time. It also allows one to explore more than one sequence of optimizations, and compare the results. The third advantage of remaining within the source language is that the programmer (and in the future, the system itself), can more easily verify that the optimizations are correct. As I will discuss later, if one limits optimizations to those which are "completely correct" for all cases, there are very few of them in real programs operating on real data.

I.2 Outline of Thesis

The bulk of the thesis is devoted to examples of how the system works. Chapter II describes the various pieces of the system, and how they fit together. Included are descriptions of how the system manipulates the program, the library of static knowledge, and the tools used to investigate the dynamic execution characteristics of the program.

Chapter III is devoted entirely to examples, starting with a series of standard Lisp functions, and how they can be improved. The number of possibilities for even the simplest of functions is surprising. Most of the improvement to the standard functions comes from transforming the function from a recursive to iterative form. The rest of chapter III discusses the results of using the system to optimize some large programs. These are included to show that the system can deal with real programs by treating program forms that it doesn't know about explicitly as "black boxes."

The last chapter contains conclusions, an assessment of the system, and suggestions for future work. An introduction to the dialect of Lisp used in the system is included as an appendix.

II

ORGANIZATION OF SYSTEM

This chapter begins with a discussion of how the choice of language influences the construction of a program transformation system. Then the programmable language-oriented editor used by the system to carry out all program changes is described. The next section covers the pattern library, where pairs of input-output patterns are stored; these pattern-pairs implement most of the optimizations carried out by the system. The next two sections describe programs which are used to analyze both the static and dynamic characteristics of a program, and which can be called by other parts of the system. The last section in the chapter discusses the question of program testing versus program proving, and gives examples to show why limiting the system to "perfectly correct" transformations is not a good idea.

II.1 The Choice of Language

When building a program transformation system, the choice of what language to use is very important. There are a number of features which a programming language should have to make the job as easy as possible -- and the result as useful as possible.

II.1.a Easy to Manipulate?

The language should be easily manipulable. This implies that the language be modular and have a consistent and simple syntax. For most infix operator Algol-like languages, this would mean that as a first step one would have to parse the input program into a tree structure so that constructs such as function calls, loops, and conditional clauses could be recognized and edited by the system. On output, the tree structure would have to traversed to regenerate the source code. To be easily manipulable also implies that the implementation of the language be as free of restrictions as possible. For example, if the language does not allow certain constructs in the arguments of a function call, or there is some arbitrary restriction on the amount of embedding of constructs, then the manipulation of that language will be much more difficult than if all constructs are allowed anywhere to any complexity.

II.1.b Sufficiently Powerful?

The language must be powerful enough to be useful for practical purposes. Optimization would be much easier in a "theoretical" language where one did not have to worry about the scope of variables, the assignment statement, and destructive data structure operations. Such languages are very interesting, but are rarely used for production programming. Another reason for language power is that without it, many optimizations will not be expressible in the original language. Also, the more power, the more likely we are to find powerful optimizations: a language with just iteration, or just recursion, will not encourage us to "think of the possibilities."

II.1.c Simple?

Somewhat at odds with the need for powerful constructs is the need for the language to be reasonably simple to understand. Simplicity is necessary so the programmer can understand the entire language, and also so the automatic optimization system can understand it. This desire for simplicity may or may not argue for extensible languages (like ECL [Cheatham 76]), since it is very difficult to get a program to understand the concept of extensibility well enough to generate optimization strategies for the extended elements of the language. Since compilers do exist for extensible languages, extensibility is clearly understood well enough for the purposes of compiling, but this is certainly not enough.

II.1.d Lisp

I chose a dialect of the Lisp language because I felt it had most of these features and because I have had long experience using it. I also have written my own interpreter for the language on the DEC PDP-10 computer, and this allows me to control the implementation of new features and the removal of bad ones. This factor was important, since many languages are either so standardized or so complex that modifications to the language definition are difficult. For example, one important feature of this Lisp, uniform handling of exceptions, (described in [Rutter 77]), is crucially important to the ability for controlled execution -- this feature would have been difficult to implement in a "standard" language, or even one of the standard PDP-10 Lisps [Moon] [Quam] [Teitelman].

The syntax of Lisp makes it easy to get started on a manipulation system -- the only parsing required is done by the READ function, which is easily understood. Similarly, getting the modified program back out in Lisp format is also easy. Since the language is interpreted directly off the program tree, and the tree appears in memory in exactly the same format as it does on paper, the problem of dealing with dual representations of the program does not arise. While it is not difficult to parse Algol-like languages into a program tree, I believe it would have proved quite an impediment to have to think about "how this text fragment appears in tree format." (I feel compelled to defend the choice of Lisp because it has the reputation of being difficult to

understand, an AI cult language, and generally "just a mass of parenthesis." Personally, I find the current trend in "structured languages" of carrying the BEGIN-END syndrome to its ultimate conclusion (IF/FI, CASE/ESAC, DO/OD, PROC/CORP, etc) far more confusing than using only one set of bracketing symbols.)

The version of Lisp used as a manipulation language has proved reasonably powerful -- the entire manipulation system was written in it. A modified compiler for the Lisp 1.6 system [Quam] serves as a compiler for a subset of the language -- this was used only when absolutely necessary for speed reasons, which turned out to be relatively seldom.

The language and syntax are briefly described in the appendix.

II.2 Programmable Language Oriented Editor

A major component of any program transformation system is an editor which actually carries out the changes. The character-oriented or line-oriented editors normally available for use on computer systems are very poorly suited to this task. Not only are most text editors not programmable to any significant extent, but they also must be run as separate, stand alone jobs. What is needed is an editor which deals with program "pieces" as its standard unit. For example, the editor needs to have the same syntactic definition of "symbol" as the language does, and moving from one symbol to the next should be an elementary operation. It is also necessary to be able to recognize higher level constructs in the language. Conditional clauses, function calls, loops, functions definitions, etc. should all be recognizable by the editor.

In the rest of this section, I will briefly describe the Lisp oriented editor used by the transformation system to carry out all the low level manipulations of programs.

II.2.a Basic Capabilities

The editor used is actually an augmented version of an interactive Lisp editor which I wrote for my own use while writing and debugging Lisp code at a video terminal. Handling the program as a program is simplified by dealing exclusively with the internal tree form of the program -- no character manipulations are done. Here again the

simplicity of Lisp syntax aids us tremendously -- no lexical scan or parsing (as such) need be done -- the program is simply READ in. Once the editor has read in the program from the terminal or a file, only the internal form of the program is dealt with until the program is written back out.

The first set of basic operations needed by an editor are movement commands. Since the program is represented internally as a binary tree, the three elementary operations are: move down (to the left), move down (to the right), and move up. In order to be able to move up, one must push a "back pointer" onto a stack whenever descending in the tree; moving up then reduces to popping this stack. These elementary movement operations are too simple to be enough -- there are also single commands to get to the next element of a list, the last element, the previous element, the head of the list, the top of the entire tree being edited, and several others. The other important movement command is the "FIND" command, which searches the tree for an occurrence of a pattern. This command uses the pattern matching routine described later to find a match. In all of these movement commands, it is possible that the requested movement cannot be performed -- in this case an "editor error" is generated, which is a "soft error" in the sense that nothing happens to the tree being edited or to the editor.

The other basic capability of an editor is to make changes to the program. Elementary operations available are deletion, replacement, insertion of new items, and structure modification. Also included in

the editor are commands to replace all occurrences of a given pattern with another pattern -- these form the basis of the transformation system's static analysis portion.

II.2.b Programmability

To be useful in an automatic system, the editor must be programmable so other programs can use the editor to carry out changes. Programming the editor is done largely through command lists, which are sequences of editor commands that appear much like the sequence of commands a person would type to carry out the same task. The command lists are executed by invoking a recursive copy of the editor itself, thus command lists may be embedded to any degree. There is also a command which allows a command list to be repeated a specific number of times, or until some condition becomes true.

Some care must be given to the implementation of programmability in an editor, since the occurrence of errors in the command list should not destroy the editor or the program being edited. Indeed, I have found it useful to exploit the soft handling of editor errors by writing command lists which execute until the occurrence of some editor error (e.g. some sequence of movement commands which stops only when it can move no further).

II.2.c Pattern Matching

All the replacement and searching commands in the editor allow the specification of a pattern. The patterns can be arbitrarily complicated, and allow one to solve the problem of recognizing programming language constructs mentioned earlier. Pattern matching facilities are familiar to most Lisp programmers, since they are often given as examples in introductions to the language. They can however, be fairly tricky to debug so that they work for all the peculiar cases.

The pattern matcher that I use has six special case elements which drive it. A "?" matches exactly one thing, a "*" matches any number of things (including none). Symbols starting with the character "?" are set to the item they match, symbols starting with the character "*" are set to the list of zero or more items that they match. Lists starting with "R?" are special restrict clauses: the third element of that list is considered a restriction on the pattern -- it must evaluate to true for the pattern to match. The second element of the list is a symbol which is set to the element matched. Lists starting with "R*" are similar, but the restriction applies to the list of elements matched. Anything else appearing in a pattern must match the data exactly for the whole match to succeed.

Patterns are used not only for determining whether there is a match, but also for their side effects. The side effects are the setting of "?" and "*" variables, the setting of variables in restrict clauses, and any other side effects the programmer may have caused via a

restriction. Restrictions are the "escape mechanism" of the pattern -- since the restriction clause is simply evaluated it need not be limited to a true/false predicate, but may do anything it pleases -- including invoking the editor, asking the user questions, etc.

The simple form of "replace all occurrences of this with that" also uses a pattern notation for the code that is to be substituted. A copy of this "output pattern" is made for each replacement, with any occurrences of "?" or "*" variables replaced by their values.

II.2.d Examples

The editor command for the simple form of "replace all" is the letter "Z" (don't ask why). This command takes two arguments, an input pattern to be searched for and an output pattern to be instantiated and used as the replacement. The edit commands to apply DeMorgan's rules as a logical simplification consist of the following two command lists:

```
(Z (OR (NOT ?A) (NOT ?B))
   (NOT (AND ?A ?B)))

(Z (AND (NOT ?A) (NOT ?B))
   (NOT (OR ?A ?B)))
```

A command list to replace all occurrences of tests for EQ or equal that are guaranteed to be true with the constant "1" would be:

```
(Z ((R? ?F (MEMQ ?F '(EQ =)))
    ?A
    (R? ?B (= ?B ?A)))
   1.)
```

There is a more complicated form of "replace all" in the editor which allows finer control over the replacement process (e.g. whether or not occurrences of the pattern should be searched for in the replacement of a matched pattern). This form also allows the replacement to be the return value of a program rather than a pattern -- sometimes the desired replacement can not be conveniently specified as a simple rearrangement of the input pattern.

While designing the editor, I have tried to be very careful about making it overly complicated by building in every conceivable operation as a command. There is always an illusion of power and greatness with software that implements an enormous set of operations -- but it is usually exactly that: an illusion. Part of keeping it simple was to make it programmable. Equally important was the philosophy that, as much as possible, there should not be a separate "editor language." Since the editor is running inside a Lisp interpreter, when more powerful control constructs are needed one can simply call the interpreter from within the editor. This has been made practical by the uniform control of exceptions (errors and non-local exits) that is available in the interpreter. Thus, if from within the editor an infinitely recursive function is evaluated by mistake, the resulting stack overflow error will be caught, and control returned to the editor safely.

II.3 Pattern Library

This section describes the library of substitution patterns that is part of the transformation system. These patterns are static in that they can be used without any knowledge of the program's running behavior. The pattern library is the simplest part of the system in concept, but has turned out to be the most useful part when the system is used on real programs.

II.3.a Format of the Library

Patterns appear in the library as components of named editor command lists. The majority of the command lists are simply pattern pairs, where all occurrences of the first pattern in the program being optimized are to be replaced by the instantiation of the second pattern. Most of the pattern pairs are local patterns, in that they can be applied to a program fragment independently of program constructs outside that fragment. Some of the patterns are more complicated in that they do look at surrounding parts of the program (via restrict clauses in the pattern) to decide whether there is a match. Patterns which attempt to determine whether the return value of a computation is used are of this nature, for example.

The command lists are given names so that they can used by other programs, and appear in other command lists. Each command list deals with one set of similar program properties. This is necessary to cut

down on interference between the transformations. Higher level command lists which specify transformation strategies also appear in the library. These use the lower level lists to implement sequences of transformations. The strategy command lists also contain control structures -- so they are not simply linear lists. This allows one to decide dynamically which set of transformations to try next, and when to stop. Deciding when to stop is not as easy as it sounds since some transformations are done for the purposes of following patterns and later undone. Thus one can not simply stop cycling through the entire sequence of transformations when no replacements are made, because there will usually be some of these "noise replacements." In practice, I have found that cycling until the number of replacements per cycle settles to a constant number is a reasonably good heuristic.

More complicated formats for the pattern library were considered, but rejected due to lack of sufficient benefit. In [Loveman] the preconditions for application of the pattern that I have included as part of the pattern itself (via restrictions) are proposed as separate elements, along with post-conditions and effectiveness predicates. Another idea, also used in [Irvine], is for each pattern to have associated with it a pointer to what transformations to try next if the replacement is made. I feel that it is too bulky to do this for every pattern pair in the library. In addition, what is really needed is the capability to decide dynamically which transformation, or more likely, set of transformations, to apply next. This capability is available by

using the lower level command lists as elements of the strategy command lists.

II.3.b Transformation Sets

In this section I will briefly describe what is in some of the low level transformation sets.

The first sets of transformations applied to a program attempt to get the program into a standard form. The first step here is to get rid of synonymous functions. For example, in most Lisps, the functions NULL and NOT are identical (although not all programmers seem to realize it). The system changes all occurrences of NULL to NOT. In my Lisp, there are quite a few synonyms, some arising from historical usage, and some for typing convenience. For example, all of the following are synonyms: SETQ <-> S, DOWHILE <-> DW, DOUNTIL <-> DU, PLUS <-> +, MINUS <-> -, DIFFERENCE <-> -, ADD1 <-> 1+, GREATERP <-> >, EQUAL <-> =, and more.

The next step towards uniformity is to get rid of compound functions by expanding them to their simpler form. As a simple example, the function NEQ (for NOT-EQ) is built-in to the interpreter. It is much simpler for later transformations if occurrences of "(NEQ ?x ?y)" are replaced by "(NOT (EQ ?x ?y))", because fewer functions have to be understood. Similarly, functions which exist in the interpreter primarily to improve the readability and convenience of Lisp, such as the conditionals IF, UNLESS, ITE, AND, and OR -- are all changed to a single conditional function: COND. Some functions which take a variable

number of arguments are changed to standard formats for ease of later transformations. The function "+" for example, is defined to take zero or more arguments; with no arguments it is equivalent to 0, with one argument it is a no-op, with three arguments it is equivalent to:

(+ a b c) <-> (+ (+ a b) c)

and so on. After applying all the "uniformity transformations" all occurrences of the "+" function have two arguments -- this makes it easier to apply later arithmetic optimizations (such as: adding 0 is a no-op, and adding 1 can be replaced by the function "1+").

The problem of deciding on a standard form for programs is not a trivial one, of course. Even within the transformation system, I have found it expedient to be able to change the form of a given program construct back and forth several times. For example, it is usually easier to attempt to split a function into two functions (for the purpose of moving conditional tests out of a "loop") when the function is stated in a recursive fashion. Yet if the program is going through a second "major cycle" of transformations, it may have already been changed by the system to iterative form. Thus the ability to "go either direction" is valuable. As the system is now, only the simplest pattern pairs can be used in either direction, so complicated patterns have a separately written inversion pattern if it has been found useful. Inverses for many of the "uniformity transformations" are available for another reason: to make the modified program as readable as possible when it is finally output.

Static optimization of conditionals and logical testing is organized into several transformation sets. These rearrange conditional expressions to minimize NOTs and the number of function calls arising from embedded conditionals. Most of the system is organized around the interpreter, and optimizes for it, thus some optimizations are performed on conditional expressions that would not make any difference to a Lisp compiler. For example, on the PDP-10, every conditional jump or skip instruction is available in both complemented and uncomplemented form (i.e. if there is a "skip if x is greater than zero" instruction, there is an equally fast "skip is x is less than or equal to zero" instruction also), thus a single NOT normally has no cost as far as conditional testing goes. But in the interpreter, NOT is a function like any other, and removing a NOT by rearrangement is an improvement.

The system also does some optimization of embedded ANDs and ORs, although it avoids rearrangements that would result in duplicates of code fragments. Complete simplification of logical expressions is a very complex and costly process (the problem is NP-complete: [Hopcroft]) -- fortunately, few real programs use logical expressions that are deeply embedded, and a reasonable job can be done with just a few sets of patterns.

The conditional transformations also attempt to eliminate all the cases of useless and redundant testing -- like testing something which is always true or always false, and testing the same variable twice when there were no intervening changes to it. These "ridiculous"

optimizations should be rare in an original program, but they arise quite frequently due to other transformations, especially when a function call is replaced by the appropriately instantiated function definition (essentially, making a function into a macro).

Another area of improvement, which can be a large one for typical Lisp programs, is recursive to iterative transformation. This is the process of changing a function that calls itself into a function which uses a loop to perform an equivalent computation. This eliminates the stack manipulation overhead necessary to perform a recursive function call. The library contains a fair number of pattern pairs which carry out recursive to iterative transformation, along with some strategies that attempt to force programs into a form where they will match. The transformations were written to be as general as possible, and this often leads to further optimization of "stupid" iterative constructs, as will be seen in the examples of chapter III.

A large set of pattern pairs are devoted to introducing more efficient replacements for specific uses of general functions. Some examples of inefficient usage which occur frequently in real programs are the use of "=" where EQ is sufficient, MEMBER where MEMQ is sufficient, and (= x 0) where (ZEROP x) would be better. These sorts of transformation sets are useful by themselves as a quick way to update old programs which were written before the introduction of some new and better function into the interpreter. They can also be used to make writing a compiler for the language easier, since the compiler itself

would not have to bother with recognizing as many special cases for each function (for example, the compiler would not have to worry about recognizing the test for zero case when compiling the "=" function (which is usually a different and faster machine instruction), if a program transformation "pre-pass" has already turned all those cases into the function ZEROP).

Some incorrect usage of functions can be quite difficult to recognize. In looking at many Lisp programs, I have often seen the function MAPCAR used where the function MAPC was really intended, probably because MAPCAR is a more mnemonic name. This is particularly disastrous, because MAPCAR CONSes up a list of return values while MAPC does not; if the lists being used are very long this will lead to frequent time-consuming garbage collections. The problem is how to recognize that the program is only using the MAPCAR for its effect of mapping a function down the list, and is not using the list of returned values. If the use of MAPCAR is embedded inside many other functions which pass on but do not really use the value it returns, this can be difficult to determine. As it stands now, the system will only recognize some of the cases where MAPCAR can be replaced with MAPC. However, it would be possible to find all the possible cases by simply making the replacements one at a time and testing the program to see if it still produced the same results.

II.3.c Discussion of the Library Approach

As was mentioned earlier, the library of patterns, while the simplest part of the transformation system in concept, has turned out to be the most useful when applying the system to real, large programs. I believe that one reason for this is that it is easy to try out new ideas. Even when a pattern only serves the purpose of invoking much more complicated programs (a "predicate" which attempts to determine whether a program fragment is side-effect free, for example), it remains useful as a structuring tool. Furthermore, the pattern matching approach to program optimization seems to be much easier to extend to new areas. To extend the system to handle a large number of arithmetic simplifications only requires the addition of about 50 patterns to the library, for example. To extend most compilers to handle the same number of cases of arithmetic identities, compile time constants, and arithmetic simplifications, would undoubtedly take longer and be a more bug-prone process, because the optimizations have to be recognized and carried out within an already complicated and "spread out" program. There are limitations to the pattern matching method; one can not reasonably expect to factor polynomials with a set of patterns, for example. (Even in this extreme case, it may still be useful to invoke a polynomial factoring program from within a pattern, so that where and when it is invoked may be controlled by the placement of the pattern in the library).

The statement of optimizations as pattern pairs makes those

optimizations more readily available for other purposes. One can imagine writing a program that would try to generalize the patterns, for example. The patterns can themselves be treated as "programs" -- and possibly optimized by other sets of patterns. Another possibility would be to try and verify the correctness of the pattern pairs (more about that in the section on testing).

An important question about the method of patterns is how many are needed for a programming language with many operators? The potential problem is one of operator interdependence: if each new operator interacts with all the previous optimization patterns then the total number of patterns needed will have an exponential dependence on the number of operators available. I believe that the picture is much brighter than this in practice. If a particular language were extended to handle sets, for example, the number of new opportunities for optimization that arise are almost all concerned with the new operators on sets -- there is very little overlap with for instance, arithmetic operators, even though the sets may contain numbers. The system currently has about 190 patterns in its static transformation library. Most of these are listed in an informal working paper [Rutter 77b]. I estimate that to provide "nearly complete" coverage of all the built-in operators in the Lisp interpreter would require less than 500 patterns.

The effect of some simple pattern optimizations may someday be arrived at with more general techniques of program transformation. In [Wegbreit 76], an example of how the unnecessary CONSing involved in the

code: (APPEND (APPEND x y) z) can be eliminated by systematic rewriting is worked through by hand. The resulting specialized function gets rid of the unnecessary consing, but is not very efficient itself:

```
(DE F
    (X Y Z)
    (COND ((NULL X) (APPEND Y Z))
          (T (CONS (CAR X)
                   (F (CDR X) Y Z)))))
```

In contrast, by including the editor command list:

```
(Z (APPEND (APPEND ?A ?B) ?C)
   (APPEND ?A (APPEND ?B ?C)))
```

in the library, the example would have been translated into (APPEND x (APPEND y z)) with much less effort. Of course, the problem is that somehow that pattern pair must get into the library in the first place. My feeling is that the number of patterns is such that it is <u>not</u> too difficult for a programmer to think up and enter most of them whenever adding new operators to a language, but clearly it would be beneficial to automate this process. The fact that APPEND is associative is one of the theorems proved by the system of Boyer and Moore [Boyer] (also see [Moore] for extensions to this system), using only the definition of APPEND -- so it is definitely possible than an automatic transformation system could derive this result directly in the future. (I must point out that this entire example is a bit contrived, since most real Lisp systems allow APPEND to have any number of arguments in the first place).

In conclusion, using a large, flexible, library of patterns is probably the easiest and fastest way to build a system that does useful source-to-source optimization of programs. The main advantages are a uniform representation for programming language specific knowledge, and the ease with which new transformations can be added and experimented with.

II.4 Static Call Tracer

This section briefly describes a program called "CALL-TRACER" which examines a program fragment and outputs a data base description of all the symbols and functions which that fragment can possibly reach when executed. The data base is used by other programs to determine useful properties about the program fragment. For example, some of the patterns in the library use the function NO-SIDE-EFFECTS to determine whether the program fragment being matched is free from side-effects. This implies that the fragment does not set any non-local variables and does not call any side-effect functions like RPLACA and RPLACD (which destructively modify data structures). These properties and others can be determined from the data output by the CALL-TRACER program.

II.4.a Example of output

The function READC, which is part of the Lisp reader program to be examined in chapter III, has the following definition:

```
(DE READC
    NIL
    (COND (CFLAG (S CFLAG NIL))
          (CHARLIST (S CHAR (CAR CHARLIST)) (NEXTD CHARLIST))
          ((WHILE (MEMQ (S CHAR
                           (COND (DTYI? (DTYI))
                                 (READLIST
                                  (PROG1 (CAR READLIST)
                                         (NEXTD READLIST)))
                                 ((THROW READERROR
                                         'READLIST-ERROR))))
                        NULLS))))
    CHAR)
```

If we were examining another part of the reader program where RFADC was called, it would be useful to know something about what RFADC requires and does. The *CALL-TRACER* program, applied to the program fragment: "(READC)", would supply the following information in a form that could be used by other programs:

```
ATOMS-REFERENCED
        (CFLAG NIL CHARLIST DTYI? READLIST NULLS CHAR)
Functions-Referenced
        SUBR-FUNCTIONS
                (CAR MEMQ DTYI)
        FEXPR/MACRO-FUNCTIONS  none
        EXPR-FUNCTIONS
                (READC)
        FSUBR-FUNCTIONS
                (COND S NEXTD WHILE PROG1 THROW QUOTE)
QUOTED-THINGS
        (READLIST-ERROR)
CATCH-LABELS  none
THROW-LABELS
        (READERROR)
LAMBDA-SYMBOLS  none
        READ-WHILE-BOUND  none
        SET-WHILE-BOUND  none
Global Symbols
        READ-WHILE-NOT-BOUND
                (CFLAG CHARLIST DTYI? READLIST NULLS CHAR)

        SET-WHILE-NOT-BOUND
                (CFLAG CHAR CHARLIST READLIST)
```

In this case, we can use this information to determine, among other things, that the call to READC has side-effects (since global variables are set), that no variable binding can occur during execution (no LAMBDA-SYMBOLS encountered), that there is a THROW without an enclosing CATCH, and that no FEXPRs or MACROs are called.

Some of the information is useful primarily when used in comparison to other "call-traces." For instance, by obtaining a CALL-TRACE of two program fragments one can determine whether they can be interchanged (that is, if they commute). Sufficient conditions for commutativity are that neither fragment uses side effect functions, and neither sets a global variable that the other reads or sets.

II.4.b How it works

The call tracing program works on the internal representation of the program being optimized. It generates its data base by traversing the program fragment given it as an argument in the same order it would be traversed if evaluated. All paths of the fragment are traversed. CALL-TRACE knows about all the built-in functions which do not simply evaluate all their arguments (i.e. all FSUBRS). Thus, it knows the order of evaluation and symbol binding in the argument structure of DOWHILE, for example. Calls to functions are traced until they reach built-in functions or until a function is called recursively. When a call to a user-defined function is encountered, the definition of that function is looked up and tracing continues in the body of that

function. To determine if a symbol is used locally or globally, a list of the currently bound (local) variables is maintained during the tracing process that simulates the binding and unbinding process that the interpreter would do when evaluating the program fragment.

II.4.c Discussion

CALL-TRACE is as close as anything in my system comes to what might be termed "classical" data flow analysis (see, for example, [Allen 76] or [Graham]). The reason I have not emphasized a more traditional approach to the problems of common sub-expression elimination, live/dead analysis, and code motion is that the traditional approach is completely static -- it neither depends on or benefits from running the program being optimized. Due to this static analysis, the only optimizations considered are always completely "safe." Surely this is what is wanted in a production FORTRAN compiler. But in a program transformation system designed to help a programmer explore all the possibilities, limiting changes to those which are a priori safe for all programs is much too restrictive. Consider the conditions for commutativity stated above. They are sufficient conditions, but they are also very often overly restrictive conditions. By eliminating out of hand any code fragments which contain side-effect functions, we have unnecessarily eliminated a large percentage of the constructs in most real languages in use today. In actuality, occurrences of a side-effect function (like RPLACA) in either of the two fragments being considered for interchange

probably does not make the interchange illegal. Often, the only way to find out is too run the program, or simulate running the program.

As another example of the problems encountered when optimizing real languages, consider a high-level language that allows control of interrupt conditions, like PL/1. One problem here is that in performing any optimizing transformations that involve code motion, the location of interrupts (faults) may move with the code. For example, in the program fragment:

```
DO I = 1 TO N;
        ...
        X = 2;
        ...
        FOO = A(I,X);
        ...
END;
```

the assignment of x to 2 could be moved out of the loop. However, if anywhere in the context of this loop there is an ON condition:

```
ON OVERFLOW BEGIN;
        ...
        X = 0;
        Y = 0;
        ...
END;
```

then this optimization can not be done if only static information is considered. If the loop were the only thing within the scope of the ON condition (i.e. they were the only things in a block), then the programmer's intention is clear and the assignment must not be moved. The more usual case is for an ON condition to encompass a large body of

program text; the code motion transformation may work for some loops and not for others. The problem of safe optimization in languages like PL/1 is discussed in [Earnest] -- but in a traditional compiler, the decision has always been made on the safe side, thus eliminating many possibilities for optimization.

There is an alternative to overly restrictive preconditions on transformations -- program testing. Throughout my transformation system, I have taken the attitude that all transformations should be reasonable, but the list of transformations should not be limited to those that will always be correct. This attitude is strongly at odds with some others in the optimization field, who insist on more formal approaches (e.g. [Gerhart] or [Samet]). I believe however, that my way is close to how human programmers optimize programs. In addition, the combination of program testing and remaining within the source language (the programmer or automatic proof system can understand the output as easily as the input) makes this flexible approach to optimization quite justifiable. (This subject is further discussed in the section on program testing, and in the conclusion).

II.5 Dynamic Analysis Tools

This section describes several tools which are used by the system (and independently) to discover the dynamic properties of programs. By dynamic properties, I mean those which can only be observed by running (or simulating) the program on some example data. Typically these properties are various statistics about the decision points in the program, but they can be much more sophisticated (obtained, for example, by looking for patterns in a computational history of the program's execution).

Optimizing programs to take advantage of their actual usage patterns is a new area for automatic optimizers, even though simpler aspects of it are relatively obvious. The system of Low [Low], which attempts to automatically select the best underlying representation of sets for particular set manipulation programs, allows some feedback from actual runs of the program, for example. An extension of this work to a system which automatically chooses representations for associative data structures [Low & Rovner], also uses feedback from running the program (via statement counters).

Knowing something about the actual running characteristics of a program can provide valuable information to other parts of a transformation system. For example, in optimizing very large programs the transformation process may take longer than one is willing to wait. In these cases, a good approximation to the speedup afforded by

examining the whole program can usually be had by only looking at those parts of the program where most of the time is spent in a typical run. Since virtually all programs in my experience do follow the "90-10" rule (90% of the time is spent in 10% of the code), this can lead to large speedups in transformation time.

II.5.a Interpreters and Evalhook

Granted that obtaining some dynamic information about the program is desirable, the problem becomes one of how best to do it. Rough measures of the relative amount of time spent in various parts of a program can be made in a variety of ways. Perhaps the simplest "add-on" technique for many compiled languages is that of inserting statement counters. This normally amounts to having the compiler (or a preprocessor) add statements to the program which increment an element of an array when they are executed; these are inserted either before every line in the program or at the beginning of each function or procedure definition. At the end of a run of the program the array of counters can be used to print out histograms, or an augmented listing of the program with each statement preceded by the number (or percent) of times it was executed.

Another method for obtaining frequency distributions is to use hardware monitors. In this technique, the physical machine itself is monitored while it is executing the program being investigated. Usually, full monitoring is difficult, since access to internal paths of

the machine is usually limited for practical reasons. However, a satisfactory job of monitoring can be done by using a real time clock to interrupt the program at frequent intervals. Within the interrupt service routine for the clock, one can look at a "snapshot" of the program's execution -- normally the current value of the program counter is of most interest. After a run, the symbol table and load map of the program can be used along with the frequency distribution of the program counter to produce a rough table showing the percentage of time spent "in the vicinity of" the symbols (presumably labels and procedure names) of the program.

While these methods for obtaining dynamic information about programs is useful, they have obvious limitations and drawbacks. If not used carefully, they may also be misleading. In the counter-per-line method for example, the frequency count obtained does not distinguish between lines in the program which add one to something and those which may call very expensive built-in functions (or worse, hidden run-time data conversion routines) -- it is up to the user to determine cost functions. Similarly, the real time clock method may be misleading when used with an optimizing compiler which can rearrange code sequences.

The real problem with these methods is that they do not easily extend to more complicated monitoring tasks. For example, one can not easily obtain the number of times "+" was called with one of its arguments zero, or what other conditions were true when a particular monitored condition was true. All of these questions become easier to answer if the language we are dealing with can be interpreted.

Even within an interpreted language like Lisp, there are still several choices to be made about how best to implement monitoring tools. One traditional approach has been to build a TRACE facility. The basic idea here is to redefine the function of interest so that before and after the old definition is executed we can insert additional code to keep counters, check conditions, or do whatever is desired. This can be a powerful method, and since we are interpreting the language, even built-in functions like "+" can be traced. There are some difficulties with the tracing method, however. Since the functions being traced are modified, there are problems of unwanted interactions with other parts of the transformation system. First, tracing built-in functions that are used within the transformation system itself will lead to confusion and unnecessary overhead. Second, since tracing modifies the source text of the program being optimized, it will have to be "un-traced" before any pattern matching is attempted.

Another approach to providing a flexible monitoring tool is to "write your own interpreter." Since the Lisp interpreter is written in assembly language, the purpose of this method is to allow better access to the interpretation process. Because an interpreter for a subset of Lisp can be written in a page or two of Lisp code, this approach does seem appealing at first. The catch lies in the word "subset" -- to interpret the same language as the real assembly language interpreter requires a much larger and more difficult program. The problem comes from the number of built-in FSUBRs (FSUBRs do not necessarily evaluate

all their arguments, and are used primarily to implement the control structures of the language). The "interpreter simulator" must have the same definition for all the built-in FSUBRs if the entire language is to be handled correctly by the transformation system. Not only is this a duplication of the efforts of whoever wrote the original interpreter, but it can never be a complete model of a real programming system since there will always be some functions which are inherently built-in (e.g. psuedo functions which do I/O, communicate with the operating system, and do internal housekeeping like memory allocation and garbage collection).

An attractive alternative to the methods discussed is to make a small modification to the assembly language interpreter which allows most of the benefits of the simulated interpreter method, but without its costs of duplication of effort and decreased execution speed. The trick is to implement a form of software interrupt which, when enabled, occurs at the beginning of each call to the internal definition of EVAL. (This feature is also available in the MacLisp system [Moon], although the internal implementation is a bit different.) To enable the interrupt during the evaluation of a form, one calls the HOOKWITH function. HOOKWITH has two arguments, the first is the name of the interrupt function (which must have one argument), and the second is the form to be evaluated. HOOKWITH operates by modifying the internal entry point of the EVAL function in the interpreter so that the next call to EVAL will call the interrupt function with the argument to EVAL passed

as an argument to the interrupt function (the interrupt function itself must be evaluated with the interrupt off, of course). HOOKWITH then calls the newly modified EVAL to evaluate the form. The value returned by the interrupt function is used as the value of the call to eval. A complication arises because of the desire to "hook" all lower calls to EVAL that might be made while evaluating a given form. We can't simply use EVAL since the interrupt isn't enabled inside the interrupt function. Using HOOKWITH to evaluate the form won't work, due to infinite recursion. The solution is via another special built-in function: EVALHOOK. EVALHOOK takes one argument, and returns what EVAL of that argument would. However, EVALHOOK re-installs the current interrupt function into EVAL first, and then calls EVAL with the one argument in such a way that it bypasses the interrupt for the first call only -- subsequent calls to EVAL cause the interrupt, so that no calls are missed.

One advantage of this method is that we now no longer need to know the argument format and evaluation scheme of built-in functions. Built-in functions like DOWHILE need not be duplicated or fully understood by the monitoring system -- the normal assembly language definition in the interpreter is used, but every call to EVAL made from that assembly language code is now caught by the eval-interrupt which can do any arbitrary processing necessary to maintain counters, evaluate conditions, or record histories.

II.5.b COUNTCALLS

One program which uses the eval-interrupt mechanism is the COUNTCALLS program, which provides a uniform way to estimate the execution time of a program on specific data. COUNTCALLS evaluates a form with an eval-interrupt function which keeps counts of how many times atoms are evaluated, and how many times each function used in the form is evaluated. When used to compare the effectiveness of transformations to a program, the data output by COUNTCALLS gives a quick account of any changes. When used in conjunction with a table of the costs of each operation, COUNTCALLS gives an estimate of the execution time of the program. By varying the table of costs to represent the cost of operations and functions on different machines or under different evaluation strategies (e.g. interpretation vs. compilation) COUNTCALLS can be used to predict the effectiveness of different approaches.

There are some small problems with COUNTCALLS if extremely exact time estimates are needed. Again they arise from built-in functions. The problem is that the execution time of some built-in functions is data dependent. The function LENGTH for example, has a fixed overhead cost plus a variable cost based, obviously, on what the length turns out to be. Since the eval-interrupt function has access to both the form being evaluated and the eventual answer, it is easy to incorporate more accurate cost formulas into COUNTCALLS for those functions which have a variable factor. (An alternative for many functions like LENGTH is to

redefine them as interpretive functions which only call constant cost built-in functions). In the transformation system as it now exists, these variable cost factors are not implemented.

The eval-interrupt function for COUNTCALLS is not very complicated. When it is called, it determines the type of the form being passed to EVAL and increments the appropriate counter. If the form is a function call, the counter for that function is incremented. As an interesting side note, the function counters are implemented with an association list -- the original implementation was with a hash table, but the association list method is faster if the list is occasionally sorted to bring the most used functions to the front. The association list turns out to be faster even with over a hundred functions in the list, because only a handful of functions account for most of the function calls in evaluating a program.

An example of the return value of COUNTCALLS when it is monitoring a merge sorting program is given below.

```
((FORM (SORT '(1 2 3 4 5 5 4 3 2 1) '>))
 (RETURN-VALUE (5 5 4 4 3 3 2 2 1 1))
 (SYMBOL-COUNT 351)
 (FIXNUM-COUNT 0)
 (FLONUM-COUNT 0)
 (FUNCTION-COUNT 331)
 (#FUNCTIONS-CALLED 14)
 (FUNCTIONS ((CDR 99) (CAR 42)
                      (RPLACD 40)
                      (ITE 28)
                      (FUNEVAL 21)
                      (S 20)
                      (MSORT 19)
                      (DW 18)
                      (IF 13)
                      (PROG1 10)
                      (MERGE 9)
                      (CDDR 9)
                      (QUOTE 2)
                      (SORT 1))))
```

The output shows that in evaluating the given form there were 351 accesses to symbols, no accesses to numbers, and 331 function calls to a total of 14 different functions. From the counts for specific functions, it is easy to determine that over half the function calls are to just 3 functions: CDR, CAR, and RPLACD.

II.5.c BCT and Augmented Function Definitions

While the COUNTCALLS program gives good information about the overall execution costs and requirements of a program, it is desirable to have a more detailed record of execution frequencies. If a function is called in two different places in the program being monitored, then the information from COUNTCALLS does not allow one to separate out the

number of times each occurrence was executed. What is needed is some method of associating with each code fragment in the program the number of times that it was executed. Within the transformation system there is a program called BCT (for BUILD-CALL-TABLE), which builds a data structure that allows one to do exactly that.

BCT uses the eval-interrupt feature of the interpreter to gather the information needed while the program is being run. Every time EVAL is called during the execution of the program, the BCT interrupt function looks up the form being passed to EVAL in a hash table. If it is not there, it adds it to the table, initializing an associated count to 1. If it is there, the count is incremented. Occurrences of the same function call in different places in the program are kept separate by hashing on EQ instead of EQUAL. This is slightly hackish, but works fine with this Lisp system because function definitions are never moved around in memory (non-compacting garbage collector) -- so hashing on the numerical value of a pointer is safe.

At the end of running the program, the hash table can be used by other programs to generate other data structures or answer questions about the program's execution. One set of optimizing transformations which is facilitated by using BCT is that of rearranging the order of conditional tests. In the simplest case, this amounts to noticing that in an expression like:

(OR a b c d)

where expressions a, b, c, and d are all side-effect free and have the same execution cost -- the tests should be reordered so that the one most likely to be true is tested first (remember, the Lisp OR evaluates left to right just until it finds a true item).

To facilitate optimization of conditional tests of all kinds, it was found desirable to have BCT run a slightly modified version of the program. This was due to a desire to easily calculate the percentage of time a test in a conditional statement is true. A problem arises because BCT does not keep track of references to atoms (it can't, since all symbols in Lisp are "interned," and if a symbol occurred in two different places in the program it would hash to the same location), and because the conditional statement may be degenerate (nothing following the test). To get around these problems, BCT uses the editor to introduce enclosing dummy function calls at those points in the program which would otherwise be ambiguous. Since at this point the program has already been transformed into the standard format, the only places that have to be changed are in the test position of the "arms" of COND, and in the arms of the SELECT function (case statement).

The hash table built by BCT can be used by other programs in a variety of ways. It can be used directly by patterns that are stored in the library, for example. Things which are of particular interest to look for are conditional predicates which are always true or always false, and code which is never executed. Potentially interesting is a

predicate in a compound conditional which evaluated to true only once during the run (when the entire conditional was evaluated many times) -- this predicate is a prime candidate to try to move "out of the loop" since it probably deals with a starting or ending condition which may not have to be tested every time.

Many of these cases become clearer if one deals with an augmented function definition instead of the original definition and a separate hash table of counts. A program is available to generate these augmented definitions from the original. It works by traversing the original function definition to make a copy of it -- but substituting a form prefaced by a number or pair of numbers for those forms that are in the hash table. A single number precedes most forms -- it is the number of times that form was executed. A pair of numbers precedes those forms which are used as predicates in conditional statements (those which were enclosed in a special dummy function before the program was run by BCT) -- the first of the pair is the number of executions, the second is the number of times that form evaluated to non-NIL (i.e. "true"). The dummy function is removed from the augmented copy. The special dummy functions inserted in the arms of the SELECT function are replaced by a number which is the number of times that arm of the SELECT was taken.

The augmented function definition retains the same tree structure as the original, so it is easy to apply patterns from the library in the same fashion as usual. If the augmented definition is to be printed out for the programmer to look at, it is a good idea to use the "reversal

patterns" in the library to make the program as readable as possible. Of particular importance to readability is the reversal of AND and OR conditional statements.

An imaginary function FOO and an augmented function definition for it are shown below.

Original Definition

```
(DE FOO
    (X Y)
    (COND ((> X 2.) (BAR))
          ((< X -3.) (ZOT))
          ((OR (GOO Y) (GOO2 Y)) (BAZ))
          (T (SELECT Y
                     (HAT (S1))
                     (GLOVES (S2))
                     (SHOES (S3))
                     (OTHERWISE)))))
```

Augmented Definition

```
(DE FOO
    (X Y)
    (100. (COND (((100. 20.) (> X 2.)) (20. (BAR)))
                (((80. 40.) (< X -3.)) (40. (ZOT)))
                ((40. (OR ((40. 5.) (GOO Y)) ((35. 15.) (GOO2 Y))))
                 (20. (BAZ)))
                (T (20. (SELECT Y
                                (HAT 4. (S1))
                                (GLOVES 10. (S2))
                                (SHOES 2. (S3))
                                (4. (OTHERWISE)))))))))
```

From the augmented definition it is easy to see how to reorder the conditional tests and the elements of the SELECT function to take

advantage of the statistical distributions in the data. Naturally, it is important to choose "typical" data to run the program on when making these sorts of optimizations. In more complex programs, the problem of reordering tests is further complicated by the occurrence of side-effects in the tests (inhibiting any rearrangement) and the differing execution costs of tests (so that cost times frequency must be used to determine total cost, rather than just frequency).

II.6 Program Testing

Another tool which relies on the ability of the system to run the program is the program tester. Program testing is used by the system to give some assurance that the transformed program gives the same answers as the original program. I am well aware of the heated battle between those who favor program testing and those who insist on program proving. It is the intent of this section to explain why I believe program testing must be used for practical programs written in real languages.

II.6.a Program Proving

The problem with proofs is not that they are inherently bad, but rather how to obtain useful proofs for real programs. Part of the difficulty is that theorem proving is a difficult activity for both man and machine. A more serious limitation arises in the requirement of most proof approaches for programmer produced assertions -- just the function definition is not enough. A program "proof" then attempts to show that the program satisfies its assertions. There are two problems here. The first is that to be useful and safe, the assertions must be free of bugs -- presumably what we want to prove about the program itself. In a large program, the generation of these assertions can be very difficult and burdensome (there has been work on automating some of the assertion generation process -- see for example, [Wegbreit 74] or [Katz]). The second problem is that we very often do not know exactly

what it is we want to prove about some aspect of the program, or how to state it as an assertion if we do know. This results in simply leaving out assertions about many parts of a large program. This problem is a fundamental difficulty with proofs of correctness. The assertions needed with a program typically involve some sort of description of the "input/output constraints" of the program. Consider the difficulty (impossibility) of specifying the input/output constraints for programs like compilers or operating systems. The likelihood of making an error (especially an error of omission) in specifying these constraints in terms of a formal assertion language is as least as great as that of the program itself containing an error.

If the current outlook for programming proving is so bleak, what is the alternative? Unfortunately, there is no alternative to a formal proof if correctness is what we must have. I suggest however, that people have gotten along remarkably well with "incorrect programs" for some time now, and promise to do so in the future. This is not intended to be a flippant remark -- it may well be that correctness proofs of large programs will always remain beyond our practical, time-bounded reach, and we will have to be satisfied with programs that are "almost certainly" correct. I believe there is an interesting parallel here with the technique of showing that very large numbers are "probably prime." For an interesting discussion of whether this is an acceptable mathematical notion, see [Rabin], for a discussion of similar problems with good but not perfect solutions, see [Knuth 76].

Within the transformation system, I have accepted the technique of programmer-aided program testing as the method of assuring that the transformations made are "correct." Basically, the programmer must supply a set of example inputs along with the program. The testing system can then insure that any transformed versions of the program give the same results as the original program. Of course, there is a great deal of room for error here. It is up to the programmer to select a set of examples which will exercise all the important features of the program -- this is once again a very difficult or impossible task for large programs. Nonetheless, experience leads me to believe that 95% testing is better than 95% proving. I am not alone in this belief -- for an excellent discussion of testing, including several examples of published, "proved" programs which unfortunately contain bugs, see [Goodenough]. I did not have the time to incorporate automatic aids to the generation of test data into the system, but some aids would be relatively easy to add. One aid is to allow generator functions in the specification of example inputs. Thus the programmer might specify that an argument to a function takes a list as an argument, and give a generator function for typical elements. This system could then automatically test the function for the most obvious "special cases" of lists: the empty list, list of one element, list of even length, list of odd length, and "long" list. Similarly, if a function takes a numerical argument, the system could generate test cases that would include: zero, plus and minus one, and the largest and smallest numbers

representable on the machine. More sophisticated automatic generation of test data can be done by looking at the control structures in the program, and attempting to generate data which will exercise all the paths. This approach to program testing is further discussed in [Huang].

There are also several practical problems involved with testing a program. One difficulty is with deciding if two versions of a program really do do the same thing. With a simple function that just returns some computed value this is easy: just compare the returned values. With more complex programs, there will probably be global side-effects either in the form of setting global variables, modifying global data structures, or doing I/O. The problem can be very complicated: if one version of the program writes to files in a different order than the other version, but at the end the two files have the same contents, do the two programs do the same thing? Also confusing the issue is that many practical programs have "unused" side-effects (or return values) -- it may be of no importance what is in a "temporary" file or internal data structure at the end of execution.

I have tried to point out some of the problems with the proof approach and with the testing approach to program correctness. My belief is that testing is the more fruitful approach. The system that is actually built uses testing, but my programming effort in the system has been elsewhere, so many of the problems are currently "left to the user." This is unfortunate, but has not proved an impediment to the

actual use of the system, which is highly interactive. Once again, manipulating the program within the source language means that the programmer can understand the results of a transformation, and intercede in the optimization process if necessary.

For further discussion of the controversy surrounding the proof vs. testing debate, see [Tanenbaum], which also includes a bibliography of program testing.

II.6.b Correct vs. Likely Optimizations

In constructing an optimizing compiler, the attempt is to only perform optimizations that are guaranteed to be safe. In constructing this transformation system, I have explicitly tried to default "the other way" to allow the system to explore many more possibilities. In real languages running on real machines, there are surprisingly few transformations which can be guaranteed correct under all circumstances. The purpose of testing is to limit the use of transformations which have a good chance of working to those instances where they actually do work.

Even some very simple "optimizations" do not actually work in all cases. Consider for example the transformation:

```
(NOT (NOT ?x))  -->  ?x
```

This is "correct" in Lisp only if the expression ?x can only take on the values "T" and NIL -- in general the expression will be either NIL or non-NIL, and the value of the expression may be used in addition to

being tested in a conditional expression. This is an example of a transformation which is "usually" correct and certainly we would want to carry it out if possible. The method used in the system is to carry it out, but back it up with testing and, if necessary, interactive questioning of the user (recall that a pattern in the library may call upon the programmer for a decision). One alternative to this approach is to ignore the possibility that the return value of the expression is being used (this transformation appears without comment in [Irvine], for example). Another possibility would be to attempt a much more complete static analysis of the values ?x can take on and the use of the value of the expression in the rest of the program. This can be very difficult, and is still subject to the pitfalls of the system not really knowing what is important (as previously mentioned, even global side-effects may not be of interest to the goals of the programmer).

As another example of an optimization that isn't always correct consider the transformation:

```
(CAR (CONS ?a ?b))  -->  ?a
```

This is used in [Wegbreit 76] for example, and in his idealized Lisp where no side-effects are possible, it is correct. But in real Lisp, the expression ?b may have side effects, and can not be eliminated out of hand. To save doing the CONS, one could first use the transformation:

(CAR (CONS ?a ?b)) --> (PROG1 ?a ?b)

and perhaps later eliminate ?b (and the then-useless PROG1). The problem of determining side-effects in a real interactive language can be very insidious. Using the static call tracer discussed earlier, one could determine whether any global variables are set and whether any side-effect functions are called, but there is a complication. In Lisp it is sometimes a useful thing to be able to "CONS up" a program on the fly and then execute it by calling EVAL. Clearly the consed-up program could have side-effects, so must we also include EVAL as a side-effect function? To be absolutely safe we must, but if we do we eliminate even more possibilities for optimization from consideration. The question of determining whether a <u>particular</u> call to EVAL has side-effects is impossible to determine statically in the general case -- what we are evaling might be something read in from a file, or even typed in from a terminal. Again, we are faced with the question of no optimization possibilities vs. optimization with the chance for error. To me, the only reasonable decision is to optimize -- with testing and interaction at the source language level the errors can be held to a minimum (the probability that the original program is strictly error-free in the first place is very, very small if the program is of any size anyway).

As a further example of optimizations which are done even though they are not strictly correct, consider the operations on numbers. It may be advantageous for a compiler to use the associativity of addition, for example:

```
(+ ?a (+ ?b ?c))  <-->  (+ (+ ?a ?b) ?c)
```

While this is correct for small integers (where there are no overflow possibilities), on real machines it is not always correct for floating point numbers. Adding a very small number to a very large number may give the same very large number unchanged. But adding two very small numbers together first before adding their sum to the large number may affect the final sum. Without knowing the values that ?a ?b and ?c can take on, one can not strictly use the above transformation -- yet most high-level language compilers do so, and some computer architectures can lead to the order of addition being indeterminate.

Yet another example of potential but unlikely trouble concerns optimizations of the unary minus operation. The following transformation is automatically applied by many compilers, for instance:

```
(- (- ?a))  -->  ?a
```

This transformation can lead to different performance of the program if the machine it is running on uses a two's complement representation for integers. The problem is that the largest negative number (-2**(n-1) where n is the word length) has no corresponding positive number. So if ?a evaluates to the largest negative number, the original program would cause an arithmetic overflow exception while the "optimized" program would not. Even the most proof oriented person would probably say that this is being too picky about correctness, because the largest negative number is so unlikely to occur -- and that is exactly my point. I

believe that the correct approach in the future is to try out transformations which are much less likely to be correct. Not only are more instances of the transformations in the library found in the program, but a "close match" of a transformation operation can be a very important piece of information to a programmer (or in the future, a more intelligent program) because it serves as a pointer to where to look in the program for further optimization.

As is clear from these examples, for the system to do a good and correct job of optimizing a program it must have more information to work with than just the program source itself (this topic is also discussed in [Newcomer pp. 23-27]). There are two good ways to get additional information about the program: ask the programmer, or ask the program (i.e. run it). The programmer can be asked interactively by the transformation system, or by requiring in the language design that additional information about variables be given in declarations (the language PASCAL is a step in this direction, allowing declaration of the range of a variable). Running the program has the advantage of least amount of work for the programmer, and also the system need not worry about the programmer inadvertantly "lying" about something in a declaration. Presumably a mixture of the two methods is the best approach.

III

EXAMPLES

The purpose of this chapter is to give a variety of examples of how the transformation system works. Examples are divided into two classes: those programs which are expressible as a single short function, and those which are at least several pages long and contain many functions.

III.1 Single Function Examples

It might be thought that programs which consist only of a single function could not be very interesting objects for optimization. With recursive Lisp functions, this is not the case. Not only can one try to convert the function to a usually more efficient iterative form, but one may also be able to minimize certain tests for "edge-effects" that seem to occur naturally from the manner in which recursive functions are written, but which actually need not necessarily be tested for on each recursive call.

The examples in this section may be considered too simple to be interesting, since they have largely been "solved" in most Lisp systems by being built-in to the interpreter. I don't think that this is the case because slight variations on these functions occur in many programs. Also, it is often of great importance to the overall optimization process that the definitions of "compound" built-in operators be available to the transformation system. This is so because many times a large part of optimization is <u>specialization</u> of functions to the particular context in which they are used. An example of this process is worked through by hand in [Loveman], where the multiplication of two matrices is optimized -- in this case, if the optimizer could not breakdown the "access to array element" operator into the hidden multiply and add operations -- no optimization would be possible.

III.1.a MEMQ

As an example of fairly simple recursive to iterative transformation, I will show the steps the system goes through when processing the function MEMQ. MEMQ looks for an instance of an element in a list, if it is in the list, the remainder of the list starting at that element is returned, if the element is not in the list, NIL is returned. The recursive form of MEMQ is very easy to write, and is similar in outline to the functions MEMBER and ASSOC in Lisp, which would be optimized in the same fashion. The definition of MEMQ which most programmers would write is:

```
(DE MEMQ
    (X L)
    (COND ((NULL L) NIL)
          ((EQ X (CAR L)) L)
          (T (MEMQ X (CDR L)))))
```

This definition is first subjected to the "uniformity transformations," which replace NULL with NOT and "T" with "1". Then attempts are made to optimize the conditional statement in the definition, but in this case, nothing can be done. Recursive to iterative transformations are tried next, and for this function, an exact match is found in the library which results in the following program:

```
(DE MEMQ
    (X L)
    (CATCH EXIT
           (DU ((X X X) (L L (CDR L)))
               ((NOT L) NIL)
               (IF (EQ X (CAR L))
                   (THROW EXIT L)))))
```

At this point another major cycle through the system is started. Now the patterns in the library relating to the optimization of do-loops can be tried, and the useless reassignment of x to itself on every pass through the loop can be removed:

```
(DE MEMQ
    (X L)
    (CATCH EXIT
           (DU ((L L (CDR L)))
               ((NOT L) NIL)
               (IF (EQ X (CAR L))
                   (THROW EXIT L)))))
```

Since there are two versions of do-loops, DOWHILE (abbreviated DW) and DOUNTIL (abbreviated DU), it is possible to remove the instance of the NOT function, and also the terminal evaluation of NIL. The resulting program is:

```
(DE MEMQ
    (X L)
    (CATCH EXIT
           (DW ((L L (CDR L)))
               (L)
               (IF (EQ X (CAR L))
                   (THROW EXIT L)))))
```

This is the final form of the function, and it runs considerably faster than the original form, as can be determined by running the program against the clock or by using the COUNTCALLS mechanism. The important thing is that the function is now an iterative one, which means that it will be faster (both interpreted and compiled) because there is no overhead in performing stack operations, and -- perhaps even

more important for this particular function -- there need be no worry about stack overflow when using MEMQ on long lists.

The actual return value of COUNTCALLS when the first and last versions of MEMQ are applied to example arguments are shown below:

Original Definition

```
((FORM (MEMQ 'X '(A B C D E F G H I J X Z)))
 (RETURN-VALUE (X Z))
 (SYMBOL-COUNT 64.)
 (FIXNUM-COUNT 0.)
 (FLONUM-COUNT 0.)
 (FUNCTION-COUNT 67.)
 (#FUNCTIONS-CALLED 7.)
 (FUNCTIONS ((COND . 11.)
             (EQ . 11.)
             (MEMQ . 11.)
             (CAR . 11.)
             (NULL . 11.)
             (CDR . 10.)
             (QUOTE . 2.))))
```

Final Version

```
((FORM (MEMQ 'X '(A B C D E F G H I J X Z)))
 (RETURN-VALUE (X Z))
 (SYMBOL-COUNT 45.)
 (FIXNUM-COUNT 0.)
 (FLONUM-COUNT 0.)
 (FUNCTION-COUNT 49.)
 (#FUNCTIONS-CALLED 9.)
 (FUNCTIONS ((IF . 11.) (CAR . 11.)
                        (EQ . 11.)
                        (CDR . 10.)
                        (QUOTE . 2.)
                        (DW . 1.)
                        (CATCH . 1.)
                        (THROW . 1.)
                        (MEMQ . 1.))))
```

III.1.b FACTORIAL

It is hard to resist transforming what is perhaps the most overused example of a recursively defined function: factorial. I start with the most popular definition.

```
(DE FACT
    (N)
    (COND ((= N 0.) 1.)
          (T (* N (FACT (- N 1.))))))
```

The first step is to take full advantage of the built-in functions available in the interpreter. The patterns that do this are some of the easiest in the library, but I have found them quite useful in updating old programs that were written for other Lisp systems or before the functions were available. In the case of FACT, the available built-in functions also make the definition smaller and more readable.

```
(DE FACT
    (N)
    (ITE (ZEROP N)
         1.
         (* N (FACT (1- N)))))
```

The second major step in transforming FACT is to try and make the definition into an iterative one. This step is preceded by some manipulations that put the conditional statements in the program into a standard format, so the number of patterns dealing with recursive to iterative transformation can be minimized. A match is found for FACT, and the resulting definition is:

```
(DE FACT
    (N)
    (DU ((N N (1- N)) (R 1. (* N R)))
        ((ZEROP N) R)))
```

This loop is certainly straightforward, and works perfectly well, but it is important to realize that the transformation to this loop depends on the multiply function "*" being associative (since in forming the factorial, the numbers are multiplied together in a different order in the iterative version than in the recursive version). Consequently, part of the pattern in the library which carries out this transformation has a restriction that the function in the position of "*" in FACT must be associative. Of course, the system does not know explicitly about the associativity of every possible function. Consistent with the philosophy of the entire system, if the associativity of the function in question is unknown, the transformation is tried anyway -- later testing will reject erroneous assumptions. Without this approach, the system would miss many chances for improvement.

III.1.c LENGTH

As another example of recursive to iterative transformation, the function LENGTH will be used. LENGTH is defined simply as follows: the length of the empty list (i.e. NIL) is zero, the length of anything longer is one plus the length of the CDR of the list. This description translates directly into the following definition:

```
(DE LENGTH
    (L)
    (COND ((NULL L) 0.)
          (T (1+ (LENGTH (CDR L))))))
```

Optimization of this function precedes as for MEMQ with uniformity transforms followed by optimizations of conditional operations. In this case a better conditional format is available, and the definition is transformed to:

```
(DE LENGTH
    (L)
    (ITE L
         (1+ (LENGTH (CDR L)))
         0.))
```

Recursion removal is now attempted, and a match is found in the library. Notice that in this case it is necessary to introduce an auxiliary variable when transforming the definition to iterative form. In general this will be necessary whenever the recursive call is embedded inside another function. The final form is:

```
(DE LENGTH
    (L)
    (DW ((L L (CDR L)) (N 0. (1+ N)))
        (L N)))
```

It is worth pointing out that this is actually what is produced by the system, in particular the actual variable name introduced in this case is "N" and not some random "G0023" name. This system is able to do this by checking to see that "N" is not already used in the program --

it has a list of likely-to-be-mnemonic names associated with the pattern in the library and tries to use one of these before resorting to generated names. It is not much more effort to do this, and in a large program one might otherwise end up with a bewildering variety of unfamiliar and totally meaningless names.

III.1.d Ith

The function Ith of two arguments finds the ith element in a list. The recursive definition is trivially developed by thinking as follows. If the list is empty, return NIL. If the element wanted is the first one, return the first element -- otherwise return the ith-1 element of CDR of the list.

```
(DE ITH
    (L I)
    (COND ((NULL L) NIL)
          ((= I 1.) (CAR L))
          (T (ITH (CDR L) (1- I)))))
```

This can be transformed into an iterative definition by the same pattern pair used for MEMQ:

```
(DE ITH
    (L I)
    (CATCH EXIT
           (DU ((L L (CDR L)) (I I (1- I)))
               ((NOT L) NIL)
               (IF (= I 1.)
                   (THROW EXIT (CAR L))))))
```

Unlike MEMQ, in this case both variables are stepped for each pass

through the loop. The remaining simple transformation is to get rid of the NOT by using DW instead of DU:

```
(DE ITH
    (L I)
    (CATCH EXIT
           (DW ((L L (CDR L)) (I I (1- I)))
               (L)
               (IF (= I 1.)
                   (THROW EXIT (CAR L))))))
```

It would appear at first that no further optimizations could be performed on this definition. There is however one further possibility for a small improvement. Since I is just being counted down and not being used elsewhere in the program, it is possible to shift the range of I so it will terminate at zero instead of one. This will usually be an improvement because testing for zero is often a special case machine instruction. Even in the interpreter the function ZEROP is available, which is faster primarily because it has only one argument to evaluate and the overhead of evaluating the constant "1" every time will be eliminated. After performing the loop shifting operation the function becomes:

```
(DE ITH
    (L I)
    (CATCH EXIT
           (DW ((L L (CDR L)) (I (1- I) (1- I)))
               (L)
               (IF (ZEROP I)
                   (THROW EXIT (CAR L))))))
```

Notice that an extra "1-" operation is done initially now to make I

start at the right value. If Ith is actually called with the first argument equal to one 99% of the time, this extra operation will make this last optimization ineffective or even of negative value. This is a case where the system implicitly assumes that loops will normally average more than one iteration. In a more "complete" transformation system of the future, this implicit assumption could be rendered harmless by testing the transformation for effectiveness by running the entire program after every small change. Currently the system does not attempt this.

III.1.e REVERSE

The function REVERSE of one argument returns a copy of the original list with the top level elements in the reverse order. In this case a good starting point is the two-function definition known to most Lisp programmers (this is as it appears in the Lisp 1.6 manual [Quam] for example):

```
(DE REVERSE
    (L)
    (REVERSE1 L NIL))

(DE REVERSE1
    (L R)
    (COND ((NULL L) R)
          (T (REVERSE1 (CDR L) (CONS (CAR L) R)))))
```

The first step in optimization is to get rid of the useless NOT in REVERSE1. This transformation has been seen in the previous examples

and may seem artificial - let me assure the reader that most people do write it that way -- presumably because it is most natural to take care of the easiest case first. Performing this transformation not only speeds up execution under the interpreter (and even with a dumb compiler), it may also increase the chances of finding a match with other patterns in later stages.

```
(DE REVERSE
    (L)
    (REVERSE1 L NIL))

(DE REVERSE1
    (L R)
    (ITE L
         (REVERSE1 (CDR L) (CONS (CAR L) R))
         R))
```

At this point we can convert directly to the DW iterative form of REVERSE1:

```
(DE REVERSE
    (L)
    (REVERSE1 L NIL))

(DE REVERSE1
    (L R)
    (DW ((L L (CDR L)) (R R (CONS (CAR L) R)))
        (L R)))
```

Nothing more can be done to REVERSE1 itself, but a closer look at what REVERSE itself does is now desirable. In the original definition REVERSE served the useful purpose of "starting" REVERSE1 with the right arguments -- the return list being built up in R has to start at NIL. But now REVERSE1 is no longer a recursive function and calls only built-

in functions. So it is possible to simply plug the definition of REVERSE1 into REVERSE with the free occurrences of its parameters replaced with actual arguments. In addition, the definition for REVERSE1 can now be removed from the program if it is not called from any other functions. The final program is:

```
(DE REVERSE
    (L)
    (DW ((L L (CDR L)) (R NIL (CONS (CAR L) R)))
        (L R)))
```

REVERSE has often been used as an example in Lisp oriented transformation systems, and some further discussion is warranted. First, there is another definition from which one might try to start optimizing -- it is conceptually the simplest, but is very inefficient since the APPEND function copies the first argument:

```
(DE REVERSE
    (L)
    (COND ((NULL L) NIL)
          (T (APPEND (REVERSE (CDR L))
                     (LIST (CAR L))))))
```

Starting with this definition (and the recursive definition of APPEND) it is very difficult to come up with the final iterative version, and the system presently can not do it. Due to the embedding of the recursive call to REVERSE inside the recursive APPEND function, it is difficult even for a person to make a series of transformations that end up with just one loop -- it is easier with an understanding of the purpose of REVERSE to simply throw the definition away and come up with

a better one. This seems to be a case with a very simple function where "first order" systems are not sufficiently powerful to make real progress. (It would of course be easy to handle this particular case by simply putting it in the library as a special case, but this is probably not particularly useful). This problem of fundamental limitations to the first order approach to problem solving is encountered in many systems (see for example, the discussion on pp. 182-185 in [Sussman]).

An aspect of REVERSE which was not considered for optimization was to make it a destructive operation. This can be done, and it can be a large savings because CONS has a delayed cost associated with garbage collection of the allocated cells. The reason REVERSE can not be changed to be destructive is that we do not know how it is going to be used in other parts of the program. Normally one does not expect a function to destroy its arguments, and if the list being reversed is pointed to by any symbol, or is part of any data structure used by the program, destructive reversal will probably have very bad effects. So the definition of REVERSE <u>itself</u> should not be made destructive (as is done in [Darlington]), but <u>particular</u> uses of it may be candidates for a destructive version of REVERSE. Those instances are best determined separately from the question of optimizing REVERSE itself -- and the problem is difficult enough that a good approach is one of brute force: replacing each instance of REVERSE with a destructive-REVERSE and testing the whole program for correctness each time.

III.1.f LAST

The function LAST, which simply returns a pointer to the last element of the list given it as argument, has more optimization possibilities than one would suspect. The starting point is the usual definition given to new students of Lisp. As is normal in any function dealing with lists, the definition starts with a check for the empty list, then checks if the current element is already the last one, otherwise the function is called recursively on the CDR of the list.

```
(DE LAST
    (L)
    (COND ((NULL L) NIL)
          ((NULL (CDR L)) L)
          (T (LAST (CDR L)))))
```

It would be possible at this point to proceed in a fashion similiar to the MEMQ example and transform the definition into an equivalent iterative form. That would be a mistake in this case however, since the iterative form would obscure the discovery of an important optimization. By first running the program for LAST on example inputs, it is possible for the system to notice a glaring "abnormality" about the first conditional test for "(NULL L)" by using an augmented function definition obtained from the BCT program described earlier. The augmented function definition will show that the test for "(NULL L)" is never true for any example input except one: the empty list itself. If the test was never true for any example it could simply be eliminated, the best we can try to do in this case is to remove it from the "main

path" of the function. The system knows of one way to attempt to do this, which is to introduce an auxiliary function definition obtained by removing all but the first test from the conditional. Testing shows that this will work in the case of LAST, and the following definition is accepted as a new starting point:

```
(DE LAST
    (L)
    (IF L (LAST1 L)))

(DE LAST1
    (L)
    (COND ((NOT (CDR L)) L)
          (1. (LAST (CDR L)))))
```

It is important to realize that the system did not arrive at this new definition by "reasoning" about the problem of finding the last element of a list, and realizing that the check for "(NULL L)" is redundant for a list element which was checked the previous time by the "(NULL (CDR L))" test. Certainly it would be nice if the system <u>did</u> have such reasoning powers, but in this case, looking for pecularities in the execution of the program has the same result.

Once the definition has been split into two functions, further optimizations of a more straight forward nature become possible. First, the conditional usage can be improved:

```
(DE LAST
    (L)
    (IF L (LAST1 L)))

(DE LAST1
    (L)
    (ITE (CDR L)
         (LAST (CDR L))
         L))
```

Now recursive to iterative transformation of the definition of LAST1 can be done directly:

```
(DE LAST
    (L)
    (IF L
        (LAST1 L)))

(DE LAST1
    (L)
    (DW ((L L (CDR L)))
        ((CDR L) L)))
```

A situation similar to one part of the REVERSE example is now apparent. The definition of LAST1 no longer contains any recursion so it may be plugged in directly where it is called in LAST -- and since it is not called anywhere else the definition may be deleted:

```
(DE LAST
    (L)
    (IF L
        (DW ((L L (CDR L)))
            ((CDR L) L))))
```

It would certainly seem that all possibilities for expressing LAST efficiently had been explored by now, since the definition is so simple.

A remaining improvement if LAST is compiled is to arrange for "(CDR L)" to be done only once for each iteration of the loop -- this should be relatively easy for a compiler that can recognize common sub-expressions. In the interpreter, it is possible to utilize the "parallel assignment" of variables in do-loops (see appendix) to obtain an unusual definition of LAST. This definition is not produced by the system, but I include it here as a matter of interest (note that the separate "outside" test for L being non-NIL is no longer necessary).

```
(DE LAST
    (L)
    (DW ((L L (CDR L))
         (L2 L L))
        (L L2)))
```

III.1.g APPEND

Another standard function that is interesting to transform is the function APPEND. As a function of two arguments, it is usually defined as follows: if the first list is empty, return the second list -- otherwise return the first element of the first list CONSed onto APPEND of the rest of the first list and the second list.

```
(DE APPEND
    (X Y)
    (COND ((NULL X) Y)
          (T (CONS (CAR X)
                   (APPEND (CDR X) Y)))))
```

As usual, it is possible to clean this up slightly by rearranging the conditional. We then have the following definition:

```
(DE APPEND
    (X Y)
    (ITE X
         (CONS (CAR X) (APPEND (CDR X) Y))
         Y))
```

The system now attempts to transform the definition into an iterative form. Although this is a very small function, it is not so easy to immediately come up with a good iterative form because of the special properties and optimizations of the CONS function. In the recursive definition of APPEND, the system stack is essentially used to reverse the order of CONSing so that the first call to APPEND returns a pointer to the first cons cell of the result. If we don't use a stack, the pointer to this first cell must be treated as a special case, and saved so it can be returned. Furthermore, to do APPEND iteratively, one must do a CONS of an element of the first list at a time when the second argument of CONS (what to cons onto) is unknown. Thus a pointer to this cons cell must be saved so that when this value is known, it can be inserted (i.e. there must be a "back-pointer" kept to the previous element of the list being copied). In addition to these special needs, the last element in the copied first list must also be treated specially, so that it is connected to the second argument. The point of all this is that making APPEND (and functions like it which use CONS) into an iterative program is a reasonably complicated task. This is the whole advantage of the library approach since we only have to figure out how to do it once -- once the pattern has been generalized and entered

into the library, the transformation system can catch all similar occurrences in the future. The library pattern in this case turns APPEND into the following iterative definition:

```
(DE APPEND
    (X Y)
    (COND (X (BINDQ ((R (LIST (CAR X))))
                    (DW ((X (CDR X) (CDR X))
                         (Y Y Y)
                         (P R (CDR (RPLACD P (LIST (CAR X))))))
                        (X (RPLACD P Y) R))))
          (1. Y)))
```

To perform this transformation, the system had to introduce two new variables. The variable R is initialized to the first cell of the result list -- its purpose is to "save this pointer." The variable P points to the previous cons cell in the iterations of the loop -- its step expression in the loop connects each copied list element to the previous end-of-list, in addition to advancing P. The function LIST, when used with one argument, is equivalent to (CONS arg NIL). Finally, when the first argument is completely copied it is connected to the second argument by the "terminal case" code: (RPLACD P Y).

The library pattern for this transformation attempts to be as general as possible, so further optimization of special cases can usually be expected. In this case, the system removes the useless initialization and setting of the variable Y to itself, and cleans up the conditional again. The final definition for APPEND is:

```
(DE APPEND
    (X Y)
    (ITE X
         (BINDQ ((R (LIST (CAR X))))
                (DW ((X (CDR X) (CDR X))
                     (P R (CDR (RPLACD P (LIST (CAR X))))))
                    (X (RPLACD P Y) R)))
         Y))
```

APPEND is unusual in that the final iterative version may not be as fast as the best recursive version running under either the interpreter or the typical dumb Lisp compiler. The reason is that the additional overhead in keeping track of back pointers and performing RPLACD's may outweigh the overhead of recursion that was removed. However, the iterative version will work on lists of any length, whereas the recursive definition will cause stack overflow for long lists (for the typical interpreter setup, lists longer than about 100 will cause overflow). Fortunately, using the COUNTCALLS program to count the number of evaluations of symbols and function calls shows the iterative version to be computationally faster than the first recursive version, although not the second. By assigning a higher than actual cost to function calls that are recursive, one can make the calculated cost favor iterative solutions and thus force the system to produce the iterative version.

III.1.h MAPCAR

MAPCAR takes two arguments, the first is a function of one

argument, the second is a list. The function is applied to each element in the list and MAPCAR returns a list of the return values. The usual definition of MAPCAR follows; the FUNEVAL function is roughly the equivalent of APPLY in most Lisps.

```
(DE MAPCAR
    (F L)
    (COND ((NULL L) NIL)
          (T (CONS (FUNEVAL F (CAR L))
                   (MAPCAR F (CDR L))))))
```

This definition can be improved in both readability and speed of interpretation by using a different choice of conditional function -- the system changes the definition to the following as a first step:

```
(DE MAPCAR
    (F L)
    (IF L
        (CONS (FUNEVAL F (CAR L))
              (MAPCAR F (CDR L)))))
```

In the second major phase of transformation, the conditional is placed in a uniform COND format again, and then recursive to iterative transformations are applied. The transformation pattern which was used for the APPEND example also matches the definition of MAPCAR. The resulting iterative form is:

```
(DE MAPCAR
    (F L)
    (COND (L (BINDQ ((R (LIST (FUNEVAL F (CAR L)))))
                    (DW ((F F F)
                         (L (CDR L) (CDR L))
                         (P R
                            (CDR (RPLACD
                                   P
                                   (LIST (FUNEVAL F (CAR L)))))))
                        (L (RPLACD P NIL) R))))
          (1. NIL)))
```

Simplifying this by removing the useless reassignment of F to itself from the loop, and changing the conditional, gives:

```
(DE MAPCAR
    (F L)
    (IF L
        (BINDQ ((R (LIST (FUNEVAL F (CAR L)))))
               (DW ((L (CDR L) (CDR L))
                    (P R
                       (CDR (RPLACD
                              P
                              (LIST (FUNEVAL F (CAR L)))))))
                   (L (RPLACD P NIL) R)))))
```

This is a reasonable definition for MAPCAR, but there is a remaining inefficiency resulting from the use of a generalized pattern in making the iterative transformation. The code to handle the "end special case," (RPLACD P NIL), is not necessary in this case. All it does is replace the CDR of a cell with NIL whose CDR is <u>already</u> NIL. While these doesn't hurt anything, and since it is outside the loop, is not particularly time wasting, it would be nice to remove it. To remove it by doing a static analysis of the situation would require the analyzer to realize two things. First, it would have to show that P

always pointed to a cons cell with NIL in its CDR. Second, it would have to know what RPLACD does, and also show that the return value of the call to RPLACD is not used. It is deceptively easy for a human Lisp programmer to "prove" these things to himself -- it is not easy to build a system to automatically do this, even in this simple case. I believe that there is a simpler way to attack this problem, which once again, relies on running the program and "listening to what the program tells you about itself." With a system similar to the BCT program described earlier, it is relatively easy to construct a special case monitor that keeps track of special cases for every usage of a function in a given program. By using the eval interrupt facility of the interpreter, one can examine the arguments passed to a function, and if throughout the execution of the program on a covering set of example inputs the function performs only useless no-op operations, the system can try to remove that instance of the function from the program. Testing the program after removal will catch cases where the return value of the function is used and thus it shouldn't have been removed. A monitor for the RPLACD function would look for instances where the CDR pointer was always being replaced with itself, which is the case in the MAPCAR definition. The concept of function monitors is only partially implemented in the system at this time. The final form of MAPCAR would be:

```
(DE MAPCAR
    (F L)
    (IF L
        (BINDQ ((R (LIST (FUNEVAL F (CAR L)))))
               (DW ((L (CDR L) (CDR L))
                    (P R
                       (CDR (RPLACD
                              P
                              (LIST (FUNEVAL F (CAR L)))))))
                   (L R)))))
```

III.1.i FIB

The patterns in the library are normally made as general purpose as possible, to increase the likelihood of application. However, some transformations that one can dream up seem to be inherently specific to very particular cases. A case in point is the pattern which changes the Fibonacci function from doubly recursive to iterative form. The pattern is generalized to handle appropriately restricted functions in the operator positions of the pattern, but it seems unlikely that any real function except Fibonacci will match the requirement that the function call itself in two places with the operation on the argument to one call being equivalent to two operations on the argument in the other call. Nonetheless, there is no harm in storing the pattern in the library, since the additional time overhead of a specific pattern like this is very small.

A starting definition for Fibonacci is given below; the important thing to notice is that the function is doubly recursive -- the execution time depends exponentially on the size of the argument.

```
(DE FIB
    (N)
    (ITE (< N 2.)
         1.
         (+ (FIB (- N 1.))
            (FIB (- N 2.)))))
```

The first step here is to translate to the standard "COND-format" and then apply the recursive to iterative transformation.

```
(DE FIB
    (N)
    (DU ((N N (- N 1.))
         (R 1. (+ R R2))
         (R2 1. R))
        ((< N 2.) R)))
```

The execution time now only depends linearly on the size of the argument, making calculation of (FIB 50) practical. The definition can be further cleaned up by using the "1-" function in place of the subtraction of one, and shifting the range of N so that the test for loop completion is faster:

```
(DE FIB
    (N)
    (DU ((N (- N 2.) (1- N))
         (R 1. (+ R R2))
         (R2 1. R))
        ((MINUSP N) R)))
```

It is worth noting that this is still not the fastest way to calculate the Fibonacci function. It is possible, by solving the recurrence relation that defines the Fibonacci function, to obtain an expression for the function which can be calculated in constant time for

any size argument (see for example, [Liu] pp. 58-62). This transformation step is currently beyond the powers of the system, but the problem of automatically solving recurrence relations has been successfully attacked in Wegbreit's METRIC "program analysis" system [Wegbreit 75], and such a capability is a natural extension.

III.2 Optimizing Large Programs

One potential difficulty with a complex optimization process is that it will take too long to be of practical value when applied to "real size" programs. This section briefly reports the results of applying portions of the transformation system to real programs.

Since the system outputs an optimized version of the original program in the source language, it would normally be used as a final pass in the development of the program, and thus need not be particularly fast. In optimizing the programs described in this section, the system took approximately two minutes of CPU time per page of source program. If the full testing and monitoring capabilities are used, this time would go up to about five minutes per page. The important thing is that the time per page does not increase dramatically with the total length of the program -- thus we do not have a "combinatorial explosion." The system itself could be made to go faster by some redesign, but this did not seem necessary at this point. The primary place where the system's speed could be increased is in the selection of patterns from the library -- the algorithm now is simply one of brute force, with many patterns applied repetitively to insure that nothing is missed. With more effort applied to this part of the system, many fewer patterns would have to be tried.

The system is not as big as one might expect -- it runs on a PDP-10 in about 60K 36-bit words. This includes the Lisp interpreter, which at about 10K, is much smaller than similar interpreters.

III.2.a Go-Moku

This program was written by the author to play the game of go-moku, or five-in-a-row. It is about 11 pages long, and is the first large Lisp program I wrote -- consequently the program is dated in both language usage and programming style. One particular thing which inhibits many possible optimizations is the use of GOTOs. At the time this program was written in Lisp 1.6, there were no looping control structures in the language except for the use of FORTRAN-style PROGs and GOs. In the language that the system was written to understand (and written in), there are neither PROGs nor GOs, these being replaced by more pleasant and structured flow of control functions. Nonetheless, the system was able to do some optimization by simply ignoring the PROGs and GOs.

Three separate parts of the program were measured against the clock to determine the amount of speedup in the optimized program. The first is the set of functions which produce a diagram of the board during play. The optimized version ran about 15% faster than the original. The second part measured was the evaluation function for the game -- this is naturally where the majority of time is spent by the program. This was only speeded up by about 10%, since the innermost function involved is completely riddled with GOTOs. The last part measured was the board expansion mechanism (the playing board automatically grows in size as the game proceeds). This part of the program benefitted substantially from the macro substitution of function bodies, and was about 25% faster than the original.

The system introduced no errors while optimizing this program, which took 25 minutes of CPU. Approximately 65 changes were made to the program, many of which were instances of improved use of specialized functions. As a result of this, the resulting program is somewhat easier to read than the original as well as being faster.

III.2.b Lisp Reader

This 5 page program is the reader for the Lisp system; it turns the character strings from a file or a terminal into internal format binary trees. I used this program to test out the parts of the system that try to reorder conditional tests according to the frequency of their actual application on real data. Much of the reader program is written in terms of case statements (the SELECT function) so this is particularly straightforward. The BCT program was used to obtain augmented function definitions for the read program, this was then used to reorder the tests so that the most frequently taken branch was tested for first. The transformation system also helped uncover an inefficiency in the most frequently called function (the read-next-character function) -- I was still testing for a particular "feature" of the reader which in fact could no longer occur.

Using the system interactively, the read program was improved to run about 15% faster. I normally compile this program, and it is interesting to note that the compiled form of the program is only about 50% slower than the carefully hand-coded and less powerful reader

AD-A041 776 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
IMPROVING PROGRAMS BY SOURCE-TO-SOURCE TRANSFORMATION.(U)
APR 77 P E RUTTER DAAB07-72-C-0259
UNCLASSIFIED R-765 NL

2 OF 2
ADA041776

END
DATE FILMED
8-77

written in assembly language for the original Lisp 1.6 system. (The original reader used jump tables to decode the characters into syntactic classes, and it is surprising that the read program written in Lisp and compiled by a poor compiler is not very much slower. The reason for recoding the Lisp reader is that the original reader had a few bugs in it, and did not allow some nice features, such as treating "2foo", "1+" and "+" as symbols instead of read errors.)

Optimizing the read program took about 10 CPU minutes; some hand guidance was needed by the system in removing the useless test once it was discovered to be useless (never executed) since that part of the transformation system has not been completed.

III.2.c Chess

This long program, about 25 pages in length, plays chess. I wrote this program about one year ago, and thus it conforms more closely to the language definition and style than the Go-Moku program mentioned earlier. Writing a chess program is a reasonably difficult task, especially if the program is to play legal chess -- that is, the program must handle all the bothersome cases like castling, en passant captures, and pawn promotion. This program does all that, and also includes interface code that provides a display on a video terminal, accepts moves, etc. I was careful when writing the program to be as efficient as possible, so I was quite interested to find out what the system could do to improve the execution speed (which is, unfortunately, too slow to be worth pursuing as a practical chess program).

Comparing the optimized program with the original, I found that, under identical conditions, playing the same 6 moves of a game took 189 seconds with the original and 154 seconds with the optimized version. This is a 22% speedup, which is much more than I expected. There were, as expected, very few instances of inappropriate function use, and no instances of recursive to iterative transformation, since I had done those by hand when the system was written. The improvement came primarily from the system automatically choosing which functions would be good candidates for making into macros, and subsequently replacing those function calls with their instantiated bodies, followed by further simplification. In the chess program, there were a large number of small functions that served as data structure accessing mechanisms -- without them the program is incomprehensible.

The transformation system took about one hour of CPU time to optimize the chess program -- far less than it would have taken me to discover and implement the same amount of speedup. Due to the macro substitution of functions the resulting program was about 10% larger than the original.

III.2.d Backplane Wiring Checker

This is a program that provides several checking and sorting facilities that are useful when making up backplane wire-wrap lists. It can check for duplicate names, order the wire-wrap list by pin name, and perform other consistency checks. The program is about 6 pages of Lisp

code; it was written several years ago by an experienced programmer and is still in use. The program has a reputation for being slow and requiring very large amounts of memory to run in when working on large backplanes. Investigation of the program showed that a good deal of time in a given run was spent garbage collecting -- and that this was due to the construction of very long lists by a poor method that copied the lists many times over.

The transformation system spent about 15 minutes of CPU time optimizing this program. Many changes were made, including those most important to cutting down on the number of garbage collections. A recursive to iterative transformation was made to the central sorting function -- something I missed when looking through the program by hand. Since this program inputs and outputs to files (something not understood by the testing program in the system) the equivalence of the output files was verified by hand.

By the clock, the optimized wire-wrap program took 55% of the time that the original took when operating on a large backplane wiring list -- in other words the improvement was almost two to one.

IV

CONCLUSIONS

The performance of the system on real programs has proved to be of practical value. While spectacular improvements in execution time are rarely obtained, the system almost always finds some improvements.

There are several reasons why some kind of optimization of programs will virtually always be necessary. One is that less than perfect programmers produce less than perfect code. Very often, especially in a language with many built-in functions, the programmer may not even be aware of the most efficient function to use for a particular case. There is also the problem of updating old programs to take advantage of new language or machine features. In both these situations the current system is at its best, and the required transformations can be carried out easily and understandably with a minimum chance for error.

Many optimization opportunities arise quite simply from poor programming. Anyone who doubts that poor programming is not the norm is encouraged to read Kernigham and Plauger's book on programming style [Kernigham]. The book gives innumerable examples of incredibly bad programs -- and they are all taken from programming textbooks!

It has been my observation that a common reason for messy and inefficient programs is that during the development process a series of editing changes to correct bugs and introduce new features leaves remnants of the "old" program around. This often leads to less than

optimal logical expressions, redundant testing, and sometimes completely useless sections of code. Part of the problem undoubtedly lies in the effort required to change the form of the program radically with a character or line-oriented text editor. If changing a conditional statement to a case statement when the program requirements indicate such a change requires many keystrokes and separate editor commands, the programmer is unlikely to make the change during the debugging process. Later, when the program is debugged, it is often too late to make the changes, since they have either been forgotten or the motivation to make them is insufficient. The use of a language-oriented editor during the program development phase can minimize the development of contorted "patched-up" code, and an automatic transformation system helps insure that the final form of the program has all the useless computation removed and the best built-in functions for each situation used.

Another advantage of having a good optimization tool is that it allows one to write clear and simple programs without undue fear of inefficiency. For example, in cases where recursion seems the most natural to the programmer, he/she can use it, realizing that the optimizer will turn it into a more efficient iterative expression if that is possible. Similarly, there is often a trade-off between the clarity of an expression and the efficiency of computing that expression. For example, in a program using sets, it may be much clearer to say:

```
(MEMBER X (UNION A B))
```

even though it will normally be faster to avoid the expensive union operation by using:

```
(OR (MEMBER X A) (MEMBER X B))
```

An optimization system can relieve the programmer of performing this chore. Using feedback from running the program, it can also order the membership tests so that the set most likely to contain x is tested first -- something few programmers would want to be bothered with doing.

Another task which is best handled by an automatic transformation system is that of specialization. This can be either specialization of the function for the particular use in the program, specialization of the program for the data, or specialization of underlying data structures (as in [Low] and [Schwartz]). Successful specialization is heavily dependent on the overall programming system design. If a "built-in" library function is to be specialized for a particular use, then its definition should be available to the optimizer in source language form -- in most current systems this is not the case.

The ability to specialize a function for a particular use is also important when the system is allowed to substitute function bodies for their calls -- in many cases a function is doing more work than is necessary for a particular call, when the body is substituted this extraneous computation can be removed.

IV.1 Assessment of System

The program transformation system developed for this thesis is of course only a beginning. Nonetheless, the more sophisticated systems of the future will, I believe, share many of its most important characteristics. Foremost among these are that transformations be carried out within one high-level language -- without this the system will be much harder to construct and making more than one pass through the system will be impossible. Getting programmers to accept a program transformation system will also be easier if they can directly see what it does.

Equally important in a transformation system is an ability to execute the program in a controlled fashion -- the "eval-interrupt" method used in this system is a big step in the right direction. Running the program not only allows the possibility of specializing the program for the data, but it also gives one a method (via specific function monitors) to discover optimizations of the program which can not be detected from the program text.

An important principle of design in the transformation system is that <u>there is no ONE all-powerful method</u> that will accomplish all of the desired goals. Thus a mixture of methods and tools must be combined in one system, with a testing system (and interaction with the programmer) to determine what is actually a correct improvement in particular cases.

In the spirit of honesty and forthrightness so well suggested in

[McDermott], I must confess to being overly ambitious in setting out to build this system. As a result, parts of the system, particularly the testing and dynamic monitoring subsystems, are incomplete and not fully debugged as of this writing. They are far from useless, but they must currently be used in a "semi-automatic mode" with the programmer maintaining the flow of control with the rest of the system, and stepping in when necessary to prevent disasters. Also, while it is possible to program the system to do any particular example for "show" purposes (like the Fibonacci function), no one should be deceived as to the limits of the power of the system. Since there is no deep model or understanding of the program being analyzed, the system can never do what is often required to get a large improvement: throw out the old version and (using what was learned) do a total rewrite. The "second order" system capable of doing this still seems many years in the future.

IV.2 Suggestions for Future Work

Many opportunities exist for extension of the transformation system. Among the possibilities are extensions to other languages, or special applications within one language (e.g. a system expert in programs dealing with a particular data structure such as sets, arrays, binary trees, etc.).

Probably the most useful thing to do is simply build up a much larger library of patterns and replacements, although this can also be a tiresome process. A good way to discover improvement patterns is to spend a lot of time reading programs written by other people (especially those just getting started in a language).

Another area for research is to automate the process of generating patterns. This can be done by generalizing from other patterns, and by proving or observing properties of functions that are useful to know when optimizing (e.g. determining the function's execution cost, whether it is associative, has side-effects, etc.).

A program transformation system can also provide useful feedback to programming language designers -- if a particular transformation is continually being done it may mean that the language does not provide a clean way of expressing what one wants to do. The metering and monitoring tools in the transformation system are very useful in designing a high level machine architecture (or compiler) for the language -- they can provide statistics as to which functions are used most often, and in what combinations.

A program transformation system can be a good influence on a programmer, showing the best way to accomplish something. A transformation system could be part of a computer assisted instruction program -- the program analysis system described in [Ruth 74] and [Ruth 76] carries out some optimizing transformations, for instance, in addition to using program transforms to aid the program recognition task.

Finally, a transformation system could be constructed to perform yet "higher-level" optimizations that would use the results of the monitoring tools. Techniques which can be tried here include the addition and deletion of program statements, the noticing of patterns and "runs" in the output of different parts of the program, and the discovery of "coincidences" between the values of variables in the program and the output. With sufficient intelligence, this can lead to the introduction of statements to exploit the previously hidden constraints on the data. Another research possibility is to write a program transformation system which could automatically introduce heuristics into the program -- here there is the additional complication that testing is now more difficult because there must be a statistical measure of "goodness" for the heuristics instead of just a right/wrong decision. These problems are discussed in more detail in [Rutter 75].

APPENDIX

A Short LISP Primer

The purpose of this appendix is to help those who are unfamiliar with Lisp. The Lisp described here is the one used in the transformation system, but the explanations of Lisp's operation have been simplified for the sake of space and sanity. A more thorough introduction to particular Lisp implementations may be found in [Moon], [Teitelman] or [Quam].

Syntax

In its simplest form, there are only three "special characters" in Lisp.

(	begins a list of items
)	terminates a list of items
space	separates items in a list

Any sequence of other characters delimited by these special characters is either a symbol or a number. The items in a list may themselves be lists; there is no limit to the amount of embedding. There is a special symbol: "NIL", which is synonymous with the list containing no items: "()". For example:

(foo bar 123 123.45e-12 (deep down) NIL)

is a list of 6 items, the first two of which are symbols, the third an

integer, the fourth a floating point number, the fifth a list of two symbols and the sixth the empty list. (For those who know Lisp: I have avoided explaining the "cons-dot" notation since it is not used in any examples).

Evaluation Rules

Evaluation of a Lisp program proceeds by passing the internal representation of the list structure to the function EVAL, which interprets its argument according to the following rules:

Type of Argument	Value Returned	
number	itself	;numbers evaluate to themselves
NIL	NIL	;NIL is also a constant
symbol	value of symbol	;all symbols have value properties
list	function call	;explained below

Lists represent function calls to EVAL -- the first element in the list is the name of the function being called and must be a symbol with a function property; the remaining elements in the list are the arguments to the function call. The function property points to the definition of the function, which is either a collection of machine instructions (in the case of built-in functions) or a list structure containing a list of the function's parameters and the definition of the function. The function property also has an indicator which gives the type of the function. There are two types of function: EXPR-type and FEXPR-type. For an EXPR-type function the arguments are evaluated left

to right and bound to the corresponding parameters in the function's parameter list. Then the body of the function definition is evaluated to obtain the value of the function call. For a FEXPR-type function the arguments are not evaluated, instead the entire argument list of the function call (i.e. everything after the function name) is bound to the FEXPR's one parameter. FEXPRs are used to implement the control structures of the language, since they allow conditional and repetitive evaluation of program fragments. Each FEXPR-type function has its own rules about the structure and interpretation of its argument.

The definitions below use the following notation:

Notation	Meaning
<sym>	there must be a symbol in this position
<exp>	any expression allowed
<pexp>	predicate-expression: value of expression used for true/false test in a conditional
<xxx>...	item preceding "..." may appear zero or more times

Some FEXPR-type functions

(DE <sym> (<sym>...) <exp> <exp>...)

DE gives an EXPR-type function property to its argument. For example:

(DE add3 (n) (+ n 3))

would define add3 to be a function which added 3 to its one argument and returned that as its value.

(IF <pexp> <exp>...)

IF evaluates its first argument, and if true (true in Lisp is anything but NIL) it evaluates its remaining arguments, returning the value of the last one as its value.

(UNLESS <pexp> <exp>...)

UNLESS is like IF except that it evaluates the remaining arguments if the first evaluates to NIL.

(ITE <pexp> <exp> <exp>...)

ITE stands for IF-THEN-ELSE. If the predicate is true it evals the second argument, otherwise the third through last.

```
(SELECT <exp>
        (<labels> <exp>...)
        ...
        <exp>)
```

SELECT is a case statement. The first expression is evaluated to get a key. The <labels> are either a single symbol or number or a list of symbols or numbers, they are not evaluated. A matching label is searched for, and if found, the expressions following it are evaluated. If no label matches, the last expression (the "default case") is evaluated.

```
(COND (<pexp> <exp>...)
      ...)
```

COND is a generalized conditional statement. The <pexp> are evaluated until one is found to be true, then the expressions following it are evaluated. COND returns NIL if no <pexp> are true.

```
(S <sym> <exp>)
```

The function S (called SETQ in most LISPs) is the assignment function: it sets the value property of the symbol to the value of the <exp>.

```
(NEXTD <sym>)
```

The function NEXTD (for NEXT-CDR) is shorthand for: (S <sym> (CDR <sym >))

```
(WHILE <pexp> <exp>...)
```

The <exp> are evaluated repeatedly as long as the <pexp> is true.

```
(UNTIL <pexp> <exp>...)
```

The <exp> are evaluated repeatedly as long as the <pexp> is NIL. The value of UNTIL is the terminal value of the <pexp>.

```
(DW (<ivar-list>...)
    (<pexp> <exp>...)
    <exp>
    ...)
```

DW stands for DO WHILE. The first list in DW's argument is a list of "iteration variable lists." Each <ivar-list>'s first element is a symbol that will be used as a local loop variable. The second element is an initial value for the variable. The optional third element is a "step-expression" that will be evaluated each time through the loop to obtain the next value of the loop variable. The second list in the argument to DW is like a single COND-arm: the loop is executed while the <pexp> is true; when it becomes NIL the <exp> following the <pexp> are evaluated to obtain the return value of the DW. The remaining arguments to DW are the body of the loop -- these <exp> are executed for each iteration of the loop. There is also a function DU (standing for DO UNTIL) which is identical to DW except that it inverts the sense of the test on the <pexp>.

Some EXPR-type functions

(LISTP <exp>)

returns true if <exp> evaluates to a CONS cell (CONS cells make up lists). The function ATOM is the inverse of LISTP. Similar type testing predicates exist for the other types: SYMBOLP, NUMBERP, FIXNUMP, and FLONUMP.

(CAR <exp>)

CAR returns the first element of a list. CAR of the list (A B C) is A. The function CDR returns the rest of the list: (B C).

(CONS <exp> <exp>)

CONS allocates a new two-element structure whose CAR is the value of the first expression and whose CDR is the value of the second.

(RPLACA <exp> <exp>)

RPLACA replaces the CAR of the cons cell pointed to by the first <exp> with the value of the second <exp>. There is a similar function RPLACD which replaces the CDR of a cell. RPLACA and RPLACD are destructive functions -- they do not allocate new structures but rather modify old ones.

(= <exp> <exp>)

returns true if the two <exp> evaluate to the same structure. = is defined recursively, two lists are = if their CARs are = and their CDRs are =. The function EQ is used to test equality of symbols.

(+ (* 3 4) (1+ 5) (- 5 2) (/ 8 2))

the value of this expression is 25. Most of the arithmetic operators can take a variable number of arguments, as + does in this example. In addition, most operators are defined for both fixed and floating point numbers -- automatic type conversion is performed where necessary.

LIST OF REFERENCES

[Allen 71] Allen, F.E., and Cocke, J., "A Catalogue of Optimizing Transformations," in Design and Optimization of Compilers, Randall Rustin, Ed., Prentice-Hall, New Jersey, 1971.

[Allen 76] Allen, F.E., and Cocke, J., "A Program Data Flow Analysis Procedure," Communications of the ACM 19,3 March 1976.

[Boyer] Boyer, R.S. and Moore, J.S., "Proving Theorems About LISP Functions," Journal of the Association for Computing Machinery, 22,1 January 1975.

[Cheatham 72] Cheatham, T.E., and Wegbreit, B., "On a Laboratory for the Study of Automatic Programming," Proceedings of the ACM Conference On Proving Assertions About Programs, January, 1972.

[Cheatham 76] Cheatham, T.E., and Townley, J.A., "A Look at Programming and Programming Systems," in Advances in Computers 14, M. Rubinoff, Ed., Academic Press, New York, 1976.

[Darlington] Darlington, J. and Burstall, R.M., "A System Which Automatically Improves Programs," Third International Joint Conference on Artificial Intelligence, pp. 479-485, Stanford University, 1973.

[Earnest] Earnest, C., "Some Topics in Code Optimization," Journal of the Association for Computing Machinery, 21,1 January 1974.

[Gerhart] Gerhart, S.L., "Correctness-Preserving Program Transformations," Second ACM Symposium on Principles of Programming Languages, January 1975.

[Goodenough] Goodenough, J.B., and Gerhart, S.L., "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, SE-1,2 June 1975.

[Graham] Graham, S.L., and Wegman, M., "A Fast And Usually Linear Algorithm for Global Flow Analysis," Journal of the Association for Computing Machinery, 23,1 January 1976.

[Hopcroft] Hopcroft, J.E., "Complexity of Computer Computations," IFIP Congress 74, North-Holland, Amsterdam, 1974.

[Huang] Huang, J.C., "An Approach to Program Testing," ACM Computing Surveys, 7,3 September 1975.

[Irvine] Standish, T.A., Harriman, D.C., Kibler, D.F., and Neighbors, J.M., The Irvine Program Transformation Catalogue, University of California at Irvine, January 1976.

[Katz] Katz, S., and Manna, Z., "Logical Analysis of Programs," Communications of the ACM, 19,4 April 1976.

[Kernigham] Kernigham, B.W., and Plauger, P.J., The Elements of Programming Style, McGraw-Hill, New York, 1974.

[Knuth 74] Knuth, D.E., "Structured Programming With Goto Statements," ACM Computing Surveys, 6,4 December 1974.

[Knuth 76] Knuth, D.E., "Mathematics and Computer Science: Coping with Finiteness," Science, 194,4271 December 17, 1976.

[Liu] Liu, C.L., Introduction to Combinatorial Mathematics, McGraw-Hill, New York, 1968.

[Loveman] Loveman, D.B., "Program Improvement by Source to Source Transformation," Third ACM Symposium on Principles of Programming Languages, January 1976.

[Low] Low, J. R., "Automatic Coding: Choice of Data Structures," Ph.D. thesis, STAN-CS-74-452, Stanford University, August 1974.

[Low & Rovner] Low, J., and Rovner, P., "Techniques for the Automatic Selection of Data Structures," Third ACM Symposium On Principles of Programming Languages, January, 1976.

[McDermott] McDermott, D., "Artificial Intelligence Meets Natural Stupidity," SIGART Newsletter, No. 57, April 1976.

[Moon] Moon, D. A., MacLisp Reference Manual, Project MAC, M.I.T., 1974.

[Moore] Moore, J.S., "Introducing Iteration into the Pure Lisp Theorem Prover," IEEE Transactions On Software Engineering, SE-1,3 September 1975.

[Nievergelt] Nievergelt, J., "On the Automatic Simplification of Computer Programs," Communications of the ACM 8,6 June 1965.

[Newcomer] Newcomer J.M., "Machine-independent Generation of Optimal Local Code," Ph.D. thesis, Carnegie-Mellon University, May 1975.

[Quam] Quam, Lynn H., LISP 1.6, Stanford Artificial Intelligence Laboratory, September, 1969.

[Rabin] Rabin, M.R., "Complexity of Computations," 1976 ACM Turing Lecture, ACM 76, Houston, Texas, October 1976. [to appear in Communications of the ACM]

[Ruth 74] Ruth, G.R., "Analysis of Algorithm Implementations," Ph.D. thesis, Massachusetts Institute of Technology, 1974.

[Ruth 76] Ruth, G.R., "Intelligent Program Analysis," Artificial Intelligence, 7,1 January 1976.

[Rutter 75] Rutter, P.E., "Optimizing Procedures: Algorithmic Improvement and Heuristic Generation," Working Paper 7, Advanced Automation Group, Coordinated Science Lab, University of Illinois, October 1975.

[Rutter 77] Rutter, P.E., "Uniform Handling of Exceptions in a Stack Based Language," to appear in SIGPLAN Notices, 1977.

[Rutter 77b] Rutter, P.E., "Source-to-Source Static Transformations for Lisp Programs," Working Paper 9, Advanced Automation Group, Coordinated Science Lab, University of Illinois, March 1977.

[Samet] Samet, H., "Automatically Proving the Correctness of Translations Involving Optimized Code," Ph.D. thesis, STAN-CS-75-498, Stanford University, May 1975.

[Schwartz] Schwartz, J.T., "Automatic Data Structure Choice in a Language of Very High Level," Communications of the ACM, 18,12 December, 1975.

[Standish] Standish, T.A., Kibler, D.F., and Neighbors, J.M., "Improving and Refining Programs by Program Manipulation," Proceedings of ACM 76, October 1976.

[Sussman] Sussman, G.J., "A Computational Model of Skill Acquisition," Ph.D. thesis, AI-TR-297, Artificial Intelligence Laboratory, M.I.T., August 1973.

[Tanenbaum] Tanenbaum, A.S., "In Defense of Program Testing or Correctness Proofs Considered Harmful," SIGPLAN Notices, 11,5 May 1976.

[Teitelman] Teitelman, W. et. al., Interlisp Reference Manual, Xerox Palo Alto Research Center, 1974.

[Wegbreit 74] Wegbreit, B., "The Synthesis of Loop Predicates," Communications of the ACM, 17,2 February 1974.

[Wegbreit 75] Wegbreit, B., "Mechanical Program Analysis," Communications of the ACM, 18,9 September 1975.

[Wegbreit 76] Wegbreit, B., "Goal-Directed Program Transformation," IEEE Transactions On Software Engineering, SE-2,2 June 1976.

VITA

Paul Edward Rutter was born in Needham, Massachusetts on September 19, 1950. He received the S.B. degree in Physical Sciences from the Massachusetts Institute of Technology in 1972.

He enrolled in the Department of Computer Science at the University of Illinois in 1972 and received the Ph.D. degree in 1977. While attending the University of Illinois he was employed as a research assistant in the Department of Computer Science and the Coordinated Science Lab.

He is a member of ACM, IEEE, and Sigma Xi.